

Resumen Teórico Programación Concurrente

Concurrencia, Paralelismo y Procesamiento Distribuido	3
Programa Concurrente, Paralelo, Distribuido y Secuencial	4
Desventajas de la Programación Concurrente	4
Clases de Aplicaciones Concurrentes	4
Paradigmas de resolución de Programas Concurrentes	5
No Determinismo, Interferencia, Comunicación	5
Sincronización y Mecanismos de Sincronización	5
Mecanismos de Comunicación y Sincronización entre procesos	6
Deadlock	7
Granularidad	7
Atomicidad de Grano Grueso y de Grano Fino	7
Sentencia Await	7
Busy Waiting - Spinning	8
Acciones atómicas condicionales e incondicionales	8
Propiedad ASV "A lo sumo una vez"	8
Propiedades de Programa, Seguridad y Vida	9
Políticas de Scheduling, Fairness y Tipos de Fairness	9
Problema de la Sección Crítica y Propiedades a cumplir	10
Algoritmos para la resolución del Problema de la Sección Crítica	10
Spin Locks	10
Algoritmo Tie-Breaker	11
Algoritmo Ticket	13
Algoritmo Bakery	13
Barreras - Sincronización Barrier	14
Barrera con Contador Compartido	14
Barrera Flag y Coordinadores	14
Barrera con Árboles - Combining Tree Barrier	15
Barrera Simétrica	16
Barrera Mariposa - Butterfly Barrier	16
Semáforos	16
Semáforo General - Counting Semaphore	16
Semáforo Binario	17
Inconvenientes de los Semáforos	17
Problemas Concurrentes y Técnicas con Semáforos	17
Problema de las Barreras - Semáforos por señalización de eventos	17
Problema de Productores y Consumidores - Semáforos binarios divididos	17
Problema de Buffers Limitados - Semáforos como contadores de recursos	18
Varios procesos compitiendo por varios recursos compartidos	19
Problema de los Filósofos - semáforos con exclusión mutua selectiva	19
Si en lugar de 5 filósofos fueran 3 ¿El problema sigue siendo de exclusión mutua selectiva?	19
Si el problema es resuelto de forma centralizada y sin posiciones fijas ¿Sigue siendo de exclusión mutua selectiva?	19
Problema Lectores y Escritores - semáforos con exclusión mutua selectiva	19

Solución como un problema de exclusión mutua	20
Solución como un problema de exclusión mutua	20
Si en el problema solo se acepta 1 escritor o un 1 lector en la base de datos ¿Tenemos un problema de exclusión mutua selectiva?	20
Técnica Passing the Baton	20
Técnica Passing the Condition	21
Relación y diferencias con Passing the Condition	21
Alocación de Recursos y Scheduling	21
Alocación Shortest-Job-Next (SJN)	22
Monitores	22
Notación de los Monitores	22
Implementación de la Sincronización por Condición	23
Protocolos de Sincronización - Disciplinas de Sincronización	23
Signal and Continue	23
Signal and Wait	23
Diferencias	24
Problemas Concurrentes y Técnicas con Monitores	24
Alocación SJN - Wait con Prioridad y Variables Condición Privadas	24
Problema del Peluquero Dormilón - Rendezvous	24
Problema de Scheduling de Disco	25
Solución con Monitor Separado	26
Solución con Monitor Intermedio	26
Conceptos Generales del Procesamiento Distribuido	26
Dualidad entre los mecanismos de pasaje de mensajes y monitores	26
Características de los Canales	27
Canal Mailbox	27
Canal Input port	27
Canal Link	27
Relación entre mecanismos de sincronización	27
Pasaje de Mensajes Asíncronos	27
Problemas Concurrentes y Técnicas con PMA	28
Productores y Consumidores - Filtros	28
Filtros	28
Solución con Red de Ordenación	28
Clientes y Servidores	29
Continuidad Conversacional	30
Pares que interactúan	30
Solución Simétrica	31
Solución Centralizada	31
Solución Anillo	31
Pasaje de Mensajes Sincrónicos	31
CSP - Lenguaje para Pasaje de Mensajes Sincrónicos	32
Comunicación guardada	32
Resultados de la evaluación de una guarda	32
Ejecución	32
Ejemplo - Criba de Eratóstenes 🧠	33

Paradigmas de Interacción entre Procesos	34
Paradigma Master/Worker	34
Paradigma Heartbeat	34
Topología de una red	34
Paradigma Pipeline	34
Paradigma Probe/Echo	34
Paradigma Broadcast	35
Paradigma Token Passing	35
Paradigma Servidores Replicados	35
RPC y Rendezvous	35
Similitudes	36
Diferencias	36
RPC (Remote Procedure Call)	36
Enfoques para proveer sincronización	37
Rendezvous	37
¿Qué elemento de la forma general de Rendezvous no se encuentra en ADA?	37
ADA - Lenguaje con Rendezvous	37
Comunicación Guardada	37
Thread	38
POSIX Threads - Pthreads	38
Funciones Básicas de Pthreads	38
Sincronización por Exclusión Mutua	38
Funciones	39
Sincronización por Condición	39
Funciones	39
Attribute Object	40
Semáforos	40
Monitores	41
Librerías para manejo de Pasaje de Mensajes	41
MPI - Message Passing Interface	41
Funciones	41
Comunicadores	42
Clasificación de Arquitecturas Paralelas	42
Arquitecturas clasificadas por Mecanismos de Control	43
SISD - Single Instruction Single Data	43
SIMD - Single Instruction Multiple Data	44
MISD - Multiple Instruction Single Data	44
MIMD - Multiple Instruction Multiple Data	44
Diseño de Algoritmos Paralelos	44
Descomposición de Datos	44
Descomposición Funcional	45
Aglomeración	45
Objetivos que guían las decisiones de Aglomeración y Replicación	45
Características de las Tareas	45
Mapeo de Procesos a Procesadores	45
Criterio para el Mapeo de Tareas a Procesadores	46

Métrica de Speedup	46
Métrica de Eficiencia	46
Programa Paralelizado	46
Ley de Amdahl	47

Concurrencia, Paralelismo y Procesamiento Distribuido

- **Concurrencia:**
 - Capacidad de **ejecutar múltiples actividades en paralelo o de forma simultánea**.
 - Es un concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.
- **Paralelismo:**
 - **Ejecución concurrente sobre diferentes procesadores**, está asociado a la existencia de múltiples procesadores ejecutando un algoritmo de forma coordinada y cooperante.
- **Procesamiento Distribuido:**
 - Conjunto de **elementos (heterogéneos) de procesamiento que se interconectan por una red de comunicaciones (pasaje de mensajes)** y cooperan entre ellos para realizar sus tareas asignadas.

Programa Concurrente, Paralelo, Distribuido y Secuencial

- **Programa Concurrente:**
 - Conjunto de **tareas o procesos secuenciales que pueden ejecutarse intercalándose en el tiempo** y que cooperan para resolver un problema.
 - Se ejecutan de forma aparentemente simultánea, pero no necesariamente en paralelo.
- **Programa Paralelo:**
 - Programa concurrente en el que los **procesos se ejecutan en paralelo** en varios procesadores.
 - El objetivo de la computación paralela es **resolver un problema** dado **más rápidamente** en pos de incrementar la performance.
- **Programa Distribuido:**
 - Programa concurrente en el que los **procesos se ejecutan en diferentes procesadores que no comparten memoria**.
 - La **comunicación** se da **por** el pasaje de **mensajes**.
- **Programa Secuencial:**
 - Programa con **único flujo de control administrado por un único procesador**.
 - Implica realizar una tarea o proceso a la vez, siguiendo un orden temporal estricto. Cada tarea debe esperar a que la anterior termine antes de comenzar su ejecución.

Desventajas de la Programación Concurrente

- **Menor Confiabilidad:**
 - Los **procesos** no son completamente independientes y **comparten recursos**. La **necesidad de utilizar mecanismos de exclusión mutua y sincronización** agrega complejidad.
- **Dificultad para la interpretación y debug:**

- Hay un **no determinismo implícito** en los procesos concurrentes. Esto significa que dos **ejecuciones del mismo programa no necesariamente son idénticas**.
- **Mayor complejidad en los compiladores y sistemas operativos asociados.**
- **Mayor costo de los ambientes y herramientas de Ingeniería de Software de sistemas concurrentes.**
- **La paralelización de algoritmos secuenciales no es un proceso directo.** Para obtener una real mejora de performance, **se requiere adaptar el software concurrente al hardware paralelo.**

Clases de Aplicaciones Concurrentes

- **Sistema de software de "multithreading"**
 - Se corresponde cuando ejecutamos **N procesos independientes sobre M procesadores, con $N > M$.**
- **Cómputo distribuido:**
 - Red de comunicaciones que **vincula procesadores diferentes sobre los que se ejecutan procesos que se comunican esencialmente por mensajes.** Cada **componente del sistema distribuido puede hacer a su vez multithreading.**
- **Procesamiento paralelo:**
 - **Resolver un problema en el menor tiempo posible,** utilizando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en diferentes procesadores.
 - Existe el **paralelismo a nivel de datos y a nivel de procesos.**

Paradigmas de resolución de Programas Concurrentes

- **Paralelismo Iterativo:**
 - Programa que tiene un conjunto de procesos cada uno con uno o más loops, es decir, **cada proceso es un programa iterativo.**
- **Paralelismo Recursivo:**
 - El problema general (programa) **puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos.**
- **Esquema Productor-Consumidor:**
 - Esquema de **procesos que se comunican,** normalmente **los procesos se organizan en pipes a través de los cuáles fluye la información.**
 - **Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el siguiente.**
- **Esquema Cliente-Servidor:**
 - Esquema **dominante en el Procesamiento Distribuido.**
 - Los **servidores esperan pedidos de servicios de múltiples clientes.**
- **Esquema de Pares que Interactúan:**
 - Los procesos (que forman parte de un programa distribuido) **resuelven partes del problema e intercambian mensajes para avanzar en la tarea.**

No Determinismo, Interferencia, Comunicación

- **No Determinismo:**
 - Los programas concurrentes son No Determinísticos, es decir, **no se puede determinar que para los mismos datos de entrada se ejecute la misma secuencia de**

instrucciones, **tampoco se puede determinar si dará la misma salida**. Sólo se puede asegurar un orden Parcial.

- **Interferencia:**

- Se da cuando **un proceso toma una acción que invalida alguna suposición hecha por otro proceso**.
- La Interferencia **se da por la realización de asignaciones en un proceso a variables compartidas que pueden afectar el comportamiento o invalidar un supuesto realizado por otro proceso**.
- Se puede **evitar usando mecanismo de sincronización y** asegurando propiedades como la de **ASV "A lo Sumo una Vez"**.

- **Comunicación:**

- **Indica el modo en que se organizan y transmiten datos entre tareas concurrentes**. Esta organización requiere **especificar protocolos para controlar el progreso y la corrección**.

Sincronización y Mecanismos de Sincronización

- **Sincronización:**

- **Conocimiento de información acerca de otro proceso para coordinar actividades**.
- Debemos considerar que:
 - **El estado de un programa concurrente** en cualquier punto de tiempo, consta de los valores de las variables de programa.
 - **El trace de una ejecución particular de un programa** concurrente puede verse como una historia.
- El rol de la sincronización es restringir las posibles historias de un programa concurrente a aquellas que son deseables.

- **Mecanismos de Sincronización:**

- **Exclusión Mutua:**
 - **Asegura que las secciones críticas** de sentencias que acceden a recursos compartidos **no se ejecuten al mismo tiempo**.
 - Conciene a **combinar las acciones atómicas de grano fino** que son implementadas directamente por hardware en secciones críticas que deben ser atómicas **para que su ejecución no sea intercalada con otras secciones críticas que referencian las mismas variables**.
- **Exclusión Mutua Selectiva:**
 - A diferencia de la Exclusión Mutua donde todos los procesos compiten contra todos por el acceso a recursos compartidos, con la exclusión mutua selectiva se **genera una competencia entre clases de procesos por el acceso a recursos compartidos**.
- **Sincronización por Condición:**
 - **Asegura que un proceso se demora si es necesario hasta que sea verdadera una condición dada**.
 - Conciene a la demora de un proceso hasta que el estado conduzca a una ejecución posterior.

Mecanismos de Comunicación y Sincronización entre procesos

- **Memoria Compartida:**

- Procesos que **intercambian mensajes sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella**.

- No se puede operar simultáneamente sobre la memoria compartida, es decir, es necesario usar algún mecanismo como los semáforos que permita bloquear y liberar el acceso a la memoria.
- Requiere el uso de exclusión mutua o sincronización por condición.
- **Memoria Distribuida:**
 - Los procesos se comunican a través de mensajes que llevan datos, es necesario establecer un canal lógico o físico para transmitir información entre procesos.
 - El pasaje de mensajes puede ser sincrónico o asincrónico y es independiente de la arquitectura.
 - Requiere del intercambio de mensajes evitando así problemas de inconsistencia.
- En un programa concurrente pueden estar presentes más de un mecanismo de sincronización a la vez.
- En términos de facilidad de programación es mucho más fácil programar con memoria distribuida porque el programador puede olvidarse de la exclusión mutua.

Deadlock

- Situación en la que dos o más procesos o hilos en un sistema concurrente quedan atrapados en un estado en el que ninguno puede continuar su ejecución debido a que cada uno está esperando a que el otro libere un recurso que necesita.
- **Condiciones necesarias y suficientes para que ocurra el deadlock:**
 - **Recursos reusables serialmente:**
 - Los procesos comparten recursos que pueden usar con exclusión mutua.
 - **Adquisición Incremental:**
 - Los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.
 - **No-preemption:**
 - Una vez que un proceso adquiere recursos, estos no pueden ser retirados de manera forzada sino que sólo son liberados voluntariamente.
 - **Espera cíclica:**
 - Existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.
- Con que evitemos que se cumpla una de estas condiciones nos alcanza para que el deadlock no exista.

Granularidad

- Es la relación que existe entre el procesamiento y la comunicación.
- **Arquitectura de grano fino:**
 - Arquitectura de muchos procesadores con poca capacidad de procesamiento.
 - Esta arquitectura es mejor para programas que requieran mucha comunicación.
- **Arquitectura de grano grueso:**
 - Arquitectura de pocos procesadores con mucha capacidad de procesamiento.
 - Esta arquitectura es mejor para programas que requieran pocos procesos que realizan mucho procesamiento y poca comunicación.

Atomicidad de Grano Grueso y de Grano Fino

- **Atomicidad de Grano Grueso:**

- Acciones que hacen una transformación de estado indivisible, es decir, cualquier estado intermedio que podría existir en la implementación de la acción no debe ser visible para los otros procesos.
- **Atomicidad de Grano Fino:**
 - Acciones que son implementadas directamente por el hardware sobre el que ejecuta el programa concurrente.

Sentencia **Await**

- Acción atómica de grano grueso, la cual es una secuencia de acciones atómicas de grano fino que aparecen como indivisibles.
- Puede ser usada para especificar acciones atómicas arbitrarias de grano grueso. Esto la hace conveniente para expresar sincronización.
- Podemos especificar sincronización a partir de usar la sentencia de la forma $\langle \text{await } (B) S; \rangle$
 - **B** es la condición de demora.
 - **S** es una secuencia de sentencias que se garantiza que termina.
 - La sentencia se encierra entre $\langle \rangle$ para indicar que es ejecutada como una acción atómica.
 - En particular, se garantiza que **B** es true cuando comienza la ejecución de **S**, y ningún estado interno de **S** es visible para los otros procesos.
- Podemos hacer solo exclusión mutua con $\langle S \rangle$.
- Podemos hacer solo sincronización por condición con $\langle \text{await } (B) \rangle$.

Busy Waiting - Spinning

- Si una sentencia **await** cumple la propiedad **ASV** puede ser implementada como *do (not B) skip od*
- Cuando se implementa la sincronización de esa forma se dice que un procesos está en **busy waiting** ya que está ocupado haciendo un chequeo de la guarda.
- Mientras se está en busy waiting se hace uso de tiempo de procesamiento del procesador.
- **Defectos de la Sincronización por Busy Waiting:**
 - Los protocolos que usan esta sincronización son complejos y no tienen una clara separación entre variables de sincronización y las usadas para computar resultados.
 - Es difícil de diseñar estos protocolos para probar la corrección del programa e incluso la verificación es compleja cuando se incrementa el número de procesos.
 - Técnica **ineficiente** si se la utiliza en multiprogramación.

Acciones atómicas condicionales e incondicionales

- **Acciones atómicas incondicionales:**
 - Acciones que se ejecutan sin considerar ninguna condición previa, estas acciones siempre se realizan de manera atómica independientemente de las circunstancias.
 - No contiene una condición de demora B. Por lo tanto, puede ejecutarse inmediatamente.
- **Acciones atómicas condicionales:**
 - Acciones que se ejecutan cuando se cumple una cierta condición previa.

Propiedad **ASV** "A lo sumo una vez"

- Una sentencia de asignación $x = e$ satisface la propiedad de "A lo sumo una vez" si:

1. "e" contiene a los uno una referencia crítica y "x" no es referenciada por otro proceso, o
 2. "e" no contiene referencias críticas, en cuyo caso "x" puede ser leída por otro proceso.
- Una expresión que no está en una sentencia de asignación satisface la propiedad de "A lo sumo una vez" si no contiene más de una referencia crítica.
 - Básicamente a lo sumo una variable compartida y a lo sumo se referencia una vez.
 - Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución parece atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

- `int x=0, y=0;` No hay ref. críticas en ningún proceso.
 `co x=x+1 // y=y+1 oc;` En todas las historias $x = 1$ e $y = 1$
- `int x = 0, y = 0;` El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
 `co x=y+1 // y=y+1 oc;` Siempre $y = 1$ y $x = 1$ o 2
- `int x = 0, y = 0;` Ninguna asignación satisface ASV.
 `co x=y+1 // y=x+1 oc;` Posibles resultados: $x = 1$ e $y = 2$ / $x = 2$ e $y = 1$
 Nunca debería ocurrir $x = 1$ e $y = 1 \rightarrow ERROR$

Propiedades de Programa, Seguridad y Vida

- **Propiedad de Programa:**
 - Atributo que es verdadero en cualquier posible historia del programa, y, por lo tanto, de todas las ejecuciones del programa.
 - Se clasifican en Propiedades de Seguridad y de Vida.
- **Propiedades de Seguridad:**
 - Aseguran que el programa nunca entre en un estado malo o inconsistente, es decir, uno en el que algunas variables tienen valores indeseables.
 - Ejemplos de esta propiedades:
 - Exclusión Mutua.
 - Ausencia de Interferencia entre procesos.
 - Partial Correctness:
 - Garantiza que si un programa termina, el resultado final es el correcto, no garantiza que el programa termine siempre.
- **Propiedades de Vida:**
 - Asegura que algo bueno eventualmente ocurre durante la ejecución, no hay deadlocks.
 - Dependen de las políticas de scheduling.
 - Ejemplos de esta propiedad:
 - Eventual Entrada.
 - Terminación:
 - Asegura que en cualquiera de las historias de ejecución del programa, el programa termine. No garantiza que el resultado final sea el correcto.
- **Caso Especial:**

- Total Correctness es la combinación de Partial Correctness y Terminación.

Políticas de Scheduling, Fairness y Tipos de Fairness

- **Política de Scheduling:**
 - Conjunto de reglas y algoritmos utilizados para determinar cuál será el próximo proceso o hilo que se ejecutará en un sistema concurrente o en un sistema operativo.
- **Fairness:**
 - Trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.
- **Fairness Incondicional:**
 - Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
- **Fairness Débil:**
 - Una política de scheduling es débilmente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece true hasta que el proceso que la hizo verdadera la vuelve falsa.
- **Fairness Fuerte:**
 - Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.
 - No aplicable en la práctica.

Problema de la Sección Crítica y Propiedades a cumplir


- Este problema consiste en desarrollar un protocolo de Entrada/Salida a la Sección Crítica para poder ejecutar una porción de código en forma atómica.
- **Propiedades a Cumplir:**
 - **Exclusión mutua (Propiedad de Seguridad):**
 - Garantiza que a lo sumo un proceso está en su Sección Crítica en un momento dado.
 - El estado "malo" a evitar es cuando múltiples procesos intentan ejecutar la sección crítica al mismo tiempo, lo que podría causar conflictos y resultados incorrectos.
 - **Ausencia de Deadlock (Propiedad de Seguridad):**
 - Garantiza que, si dos o más procesos intentan entrar en sus secciones críticas, al menos uno de ellos tendrá éxito.
 - El estado "malo" es cuando los procesos quedan bloqueados en un punto muerto y no pueden avanzar.
 - **Ausencia de Demora Innecesaria (Propiedad de Seguridad):**
 - Garantiza que si un proceso intenta entrar en su sección crítica y los otros procesos están en secciones no críticas o ya han terminado, el primer proceso no debe ser impedido de entrar en su sección crítica.
 - El estado "malo" es cuando un proceso está bloqueado a pesar de que los recursos están disponibles.
 - **Eventual Entrada (Propiedad de Vida):**
 - Garantiza que un proceso que intenta entrar en su sección crítica eventualmente lo hará.

- El estado “malo” es cuando un proceso queda excluido repetidamente de su sección crítica.

Algoritmos para la resolución del Problema de la Sección Crítica

Spin Locks

- Tiene como **objetivo hacer atómico el await de grano grueso**, para esto hacemos uso de una instrucción especial que puede usarse para implementar las acciones atómicas condicionales, en este caso **usamos Test-and-Set pero existen otras como Fetch-and-Add o Compare-and-Swap.**
- **Test-and-Set:**
 - **Toma una variable “lock” booleana como argumento.** Esta variable representa el estado del recurso compartido. Si “lock” es falso, el recurso está disponible. Si “lock” es verdadero, el recurso está siendo utilizado por otro proceso.
 - **Atómicamente, TS guarda el valor inicial de “lock” en una variable temporal:**
 - **Establece “lock” en verdadero**, indicando que el recurso ya no está disponible.
 - **Devuelve el valor original de “lock”.**
 - El proceso que llama a TS analiza el valor devuelto:
 - **Si es falso, significa que el proceso ha adquirido el recurso y puede entrar a la Sección Crítica.**
 - **Si es verdadero, significa que otro proceso ya está utilizando el recurso.** El proceso que llamó a TS debe esperar y volver a intentar adquirir el recurso.
- **Desventajas de este tipo de solución:**
 - Las soluciones de este estilo **no controlan el orden** de los procesos demorados.
 - Cumple **las 4 propiedades del problema si el scheduling es fuertemente fair.**
 - En multiprocesadores puede llevar a una **baja performance** si varios procesos están compitiendo por el acceso a una Sección Crítica, **causando Memory Contention.**
 - **Genera busy-waiting.**



```

1  bool lock=false;
2
3  process SC[i=1 to n] {
4      while (true) {
5          while (lock) skip;
6          while (TS(lock)) {
7              while (TS(lock)) skip;
8          }
9          // sección crítica;
10         lock = false;
11         // sección no crítica;
12     }
13 }

```

Solución de grano fino con Spin Locks

Algoritmo Tie-Breaker

- Protocolo de Sección Crítica que requiere solo scheduling incondicionalmente fair para satisfacer la propiedad de eventual entrada.
- No requiere instrucciones especiales del tipo Test-and-Set.
- El algoritmo es mucho más complejo que la solución con Spin Locks.
- Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada, esta última variable es compartida y de acceso protegido.
- El algoritmo tie-breaker de N-procesos es costoso en tiempo y bastante complejo y difícil de entender.



```
1  bool in1 = false, in2 = false;
2  int ultimo = 1;
3
4  process SC1 {
5      while (true) {
6          in1 = true; ultimo = 1;
7          while (in2 and ultimo == 1) skip;
8          // sección crítica;
9          in1 = false;
10         // sección no crítica;
11     }
12 }
13
14 process SC2 {
15     while (true) {
16         in2 = true; ultimo = 2;
17         while (in1 and ultimo == 2) skip;
18         // sección crítica;
19         in2 = false;
20         // sección no crítica;
21     }
22 }
```

Tie-Breaker para 2 procesos



```
1  int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);
2
3  process SC[i = 1 to n] {
4      while (true) {
5          for [j = 1 to n] {
6              # protocolo de entrada
7              # el proceso i está en la etapa j y es el último
8              in[i] = j; ultimo[j] = i;
9              for [k = 1 to n st i < k] {
10                 # espera si el proceso k está en una etapa más alta
11                 # y el proceso i fue el último en entrar a la etapa j
12                 while (in[k] ≥ in[i] and ultimo[j]=i) skip;
13             }
14         }
15         // sección crítica;
16         in[i] = 0;
17         // sección no crítica;
18     }
19 }
```

Tie-Breaker para N procesos

Algoritmo Ticket

- Solución para N procesos más fácil de entender que la solución de N procesos del algoritmo Tie-Breaker.
- En esta solución se reparten números y se espera turno:
 - Los clientes toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los clientes con un número más chico sean atendidos.
- La ausencia de deadlock y de demora innecesaria resultan de que los valores de turno son únicos. Con scheduling débilmente fair se asegura eventual entrada.
- El problema de este algoritmo es que los valores del próximo número y del turno son ilimitados, si el algoritmo corre un tiempo largo se puede alcanzar un overflow.
 - Para tratar esto podemos resetear los contadores cuando notamos que los valores son muy grandes.
- Hace uso de la instrucción especial Fetch-and-Add, sin el uso de esta instrucción se debe simular con una Sección Crítica y la solución puede no ser fair.
- **Fetch-and-Add:**
 - Esta operación hace 2 acciones atómicas:
 - Lee el valor actual de una variable.
 - Incrementa esa variable por una cantidad específica.
 - La operación devuelve el valor original de la variables (antes de incrementar) y asegura que la actualización del valor se haga de forma segura.



```
1  int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );
2
3  process SC [i: 1..n] {
4      while (true) {
5          turno[i]=FA(numero,1);
6          while (turno[i] > proximo) skip;
7          //sección crítica;
8          proximo = proximo + 1;
9          // sección no crítica;
10     }
11 }
```

Algoritmo Bakery

- Algoritmo del tipo de ticket que es fair y no requiere instrucciones de máquina especiales.
- El algoritmo es más complejo que el ticket, pero ilustra una manera de romper empates cuando dos procesos obtienen el mismo número.
- No requiere un contador global próximo que se “entrega” a cada proceso al llegar a la Sección Crítica.
- Cada proceso que trata de ingresar recorre los números de los demás y se autoasigna uno mayor. Luego espera a que su número sea el menor de los que esperan.
- Los procesos se chequean entre ellos y no contra un global.
- El algoritmo asegura entrada eventual si el scheduling es débilmente fair.

int turno[1:n] = ([n] 0);

{BAKERY: $(\forall i: 1 \leq i \leq n: (SC[i] \text{ está en su SC}) \Rightarrow (\text{turno}[i] > 0) \wedge (\forall j: 1 \leq j \leq n, j \neq i: \text{turno}[j] = 0 \vee \text{turno}[i] < \text{turno}[j]))$) }

```
process SC[i = 1 to n]
{ while (true)
  { turno[i] = 1; //indica que comenzó el protocolo de entrada
    turno[i] = max(turno[1:n]) + 1;
    for [j = 1 to n st j != i] //espera su turno
      while (turno[j] != 0) and ( (turno[i],i) > (turno[j],j) ) → skip;
    sección crítica
    turno[i] = 0;
    sección no crítica
  }
}
```

Barreras - Sincronización Barrier

- Punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución.
- Pueden reutilizarse más de una vez.


- Como muchos problemas pueden ser resueltos con algoritmos iterativos paralelos en los que cada iteración depende de los resultados de la iteración previa, es necesario proveer sincronización entre los procesos al final de cada iteración, para realizar esto se utilizan las barreras.

Barrera con Contador Compartido

- **N procesos necesitan encontrarse en una barrera:**
 - Cada proceso incrementa un contador compartido (esto lo podemos hacer con Fetch-and-Add si es posible) al llegar a la barrera.
 - Cuando el contador es N los procesos pueden pasar la barrera.
 - Opcionalmente podemos hacer que el último proceso en llegar a la barrera sea el encargado en resetear el contador para una próxima iteración.

Barrera Flag y Coordinadores

- Se tiene un proceso Coordinador que gestiona la liberación y el reseteo de la barrera para los procesos Workers que mientras van llegando van marcando su respectivo Flag de llegada.
- Reintroduce contención de memoria y es ineficiente.
- Para la mejor optimización necesitamos un procesador dedicado al proceso Coordinador.



```

1  intarribo[1:n]=([n]0),continuar[1:n]=([n]0);
2
3  processWorker[i=1ton] {
4      while(true) {
5          // código para implementar la tarea i;
6          arribo[i] = 1;
7          while (continuar[i] == 0) skip;
8          continuar[i]=0;
9      }
10 }
11
12 processCoordinador{
13     while(true) {
14         for[i=1ton] {
15             while(arribo[i] == 0) skip;
16             arribo[i]=0;
17         }
18         for[i=1ton]continuar[i]=1;
19     }
20 }

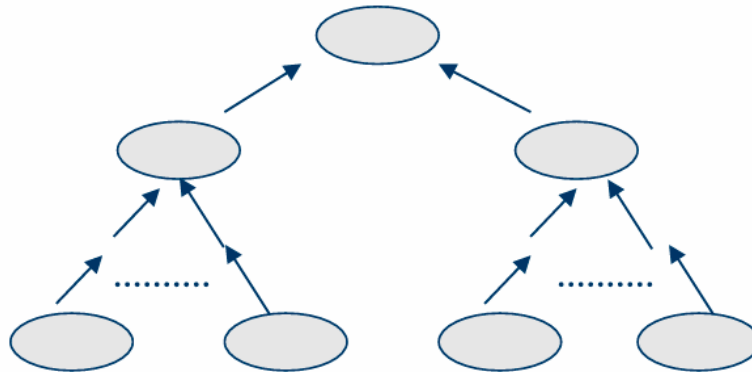
```

Barrera con Árboles - Combining Tree Barrier

- Los procesos se organizan en forma de árbol y cumplen diferentes roles.
- Los procesos envían el aviso de llegada a la barrera hacia arriba en el árbol y la señal de continuar cuando todos arribaron es enviada de arriba hacia abajo.
 - Cada proceso debe combinar los resultados de sus hijos y luego pasarlos a su padre.
- **Ventajas:**
 - Implementación sencilla.

- **Desventajas:**

- Los procesos no son simétricos ya que cada uno cumple diferentes roles, por lo que los nodos centrales realizarán más trabajo que los nodos hoja y la raíz.



Barrera Simétrica

- Conjunto de barreras entre pares de procesos que utilizan sincronización barrier. En cada etapa los pares de procesos que interactúan van cambiando dependiendo de algún criterio establecido.

Barrera Mariposa - Butterfly Barrier

- Barrera Simétrica para N procesos.
- Se hacen $\log_2(N)$ etapas siendo N la cantidad de procesos.
 - Cada Worker sincroniza con uno distinto en cada etapa.
 - Cuando cada worker pasó $\log_2(N)$ etapas, todos pueden seguir.
- **Ventajas:**
 - La implementación de sus procesos es simétrica ya que todos realizan la misma tarea en cada etapa "s" que es la de sincronizar con un par a distancia 2^{s-1} .
- **Desventajas:**
 - Implementación más compleja que la de una Combining Tree Barrier.
 - Cuando N no es par puede usarse un N próximo par para sustituir a los procesos perdidos en cada etapa, sin embargo, esa implementación no es eficiente por lo que se usa una variante llamada Dissemination Barrier.

Workers	1	2	3	4	5	6	7	8
Etapa 1	_____		_____		_____		_____	
Etapa 2	_____	_____			_____	_____		
Etapa 3	_____	_____	_____	_____				

Semáforos

- Herramienta que nos permite realizar sincronización entre procesos concurrentes. Es una instancia de un tipo de datos abstracto con sólo 2 operaciones atómicas **P y V**. Internamente el valor de un semáforo es un entero no negativo:
 - **V:**
 - Señala la ocurrencia de un evento incrementando de forma atómica el valor interno del semáforo.
 - **P:**
 - Se usa para demorar un proceso hasta que ocurra un evento decrementando de forma atómica el valor interno del semáforo.
- Permiten realizar sincronización por exclusión mutua (proteger secciones críticas) y por condición.

Semáforo General - Counting Semaphore

- $P(s): \langle \text{await } [s > 0] \ s = s - 1; \rangle$
- $V(s): \langle s = s + 1; \rangle$
- Cuando el semáforo se vuelve mayor que 0, cualquier proceso puede pasar, no respeta un orden de llegada.
 - Tenemos riesgo de que no se respete la propiedad de Eventual Entrada.
- Si la implementación de la demora por operaciones **P** se produce sobre una cola, entonces las operaciones si son fair.

Semáforo Binario

- $P(b): \langle \text{await } (b > 0) \ b = b - 1; \rangle$
- $V(b): \langle b = b + 1; \rangle$

Inconvenientes de los Semáforos

- Son variables compartidas globales a los procesos.
- Generan sentencias de control de acceso a la Sección Crítica en el código.
- Al agregar procesos, se debe verificar el acceso correcto a las variables compartidas.
- Aunque Exclusión Mutua y Sincronización por Condición son conceptos distintos, se programan de forma similar.

El problema es tener el protocolo de acceso distribuido en el código en vez de tenerlo localizado en un solo lugar.



Problemas Concurrentes y Técnicas con Semáforos

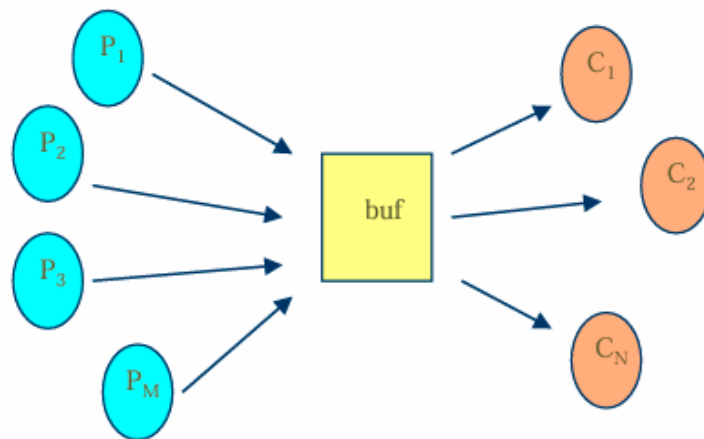
Problema de las Barreras - Semáforos por señalización de eventos

- La idea es usar un semáforo para cada flag de sincronización donde un proceso setea el flag ejecutando **V**, y espera a que un flag sea seteado y luego lo limpia ejecutando **P**.
- **Semáforo de Señalización**
 - Generalmente inicializado en 0. Un proceso señala el evento con **V(s)**; otros procesos esperan la ocurrencia del evento ejecutando **P(s)**;

Problema de Productores y Consumidores - Semáforos binarios divididos

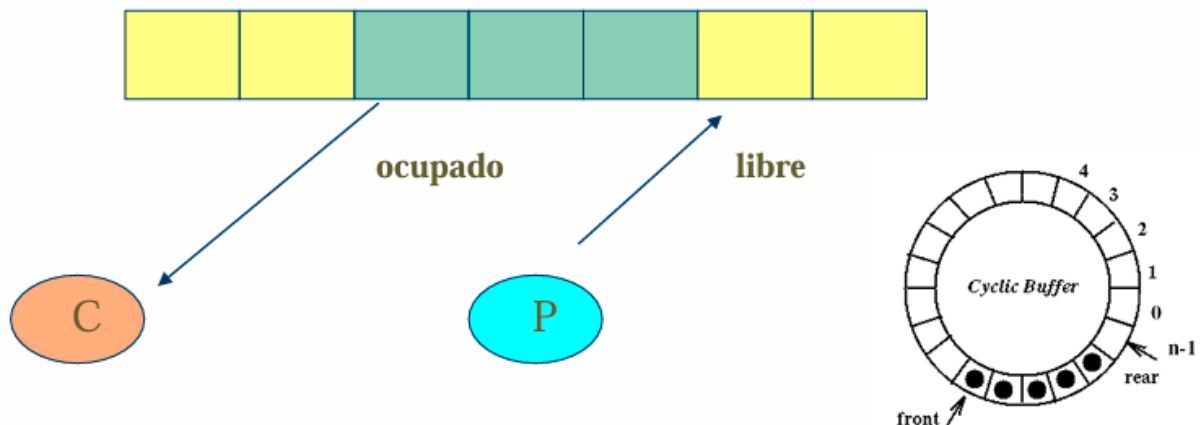
- **Semáforo Binario Dividido (Split Binary Semaphore) - SBS**

- Los semáforos binarios (en nuestro caso semáforos generales que simulan a los binarios) $b_1, b_2, b_3, \dots, b_N$ forman un SBS en un programa **si se cumple que la suma de todos los semáforos es mayor o igual que 0 y menor o igual que 1**, es decir, es binaria.
- Se puede ver como un semáforo binario que fue dividido en n partes.
- **Tienen una implementación específica para la exclusión mutua (problema de la sección crítica)**
 - En general la ejecución de los procesos inicia con un **P** sobre un semáforo y termina con un **V** sobre otro de ellos. Esto permite que se genere una alternancia entre procesos para ejecutar la sección crítica.
- **Ejemplo:** Buffer unitario compartido con **múltiples** productores y consumidores. Dos operaciones: depositar y retirar que deben alternarse.

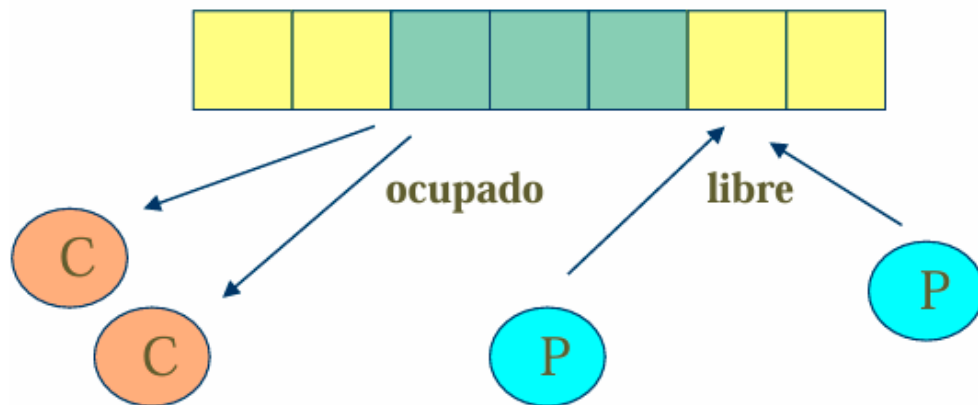


Problema de Buffers Limitados - Semáforos como contadores de recursos

- **Semáforos Contadores de Recursos**
 - **Cada semáforo cuenta el número de unidades libres de un recurso determinado.** Son útiles cuando los procesos compiten por recursos de múltiples unidades que serían, por ejemplo, un buffer con N posiciones limitadas.
- **Ejemplo:** Un buffer es una cola de mensajes depositados y aún no buscados. Existe un productor y un consumidor que depositan y retiran elementos del buffer. **Manejamos el buffer como una cola circular.**



Problema básico con un Productor y un Consumidor



Con más de un productor y un consumidor, las operaciones **depositar** y **retirar** son secciones críticas y se deben ejecutar con exclusión mutua.

Varios procesos compitiendo por varios recursos compartidos

- Hay varios procesos y varios recursos cada uno de ellos protegidos por un lock. Un proceso tiene que adquirir los locks de todos los recursos que requiere para poder trabajar, puede haber deadlock si 2 o más procesos pueden querer el mismo recurso.

Problema de los Filósofos - semáforos con exclusión mutua selectiva

- Problema que surge si varios procesos compiten por el acceso a conjuntos superpuestos de variables compartidas.
- **Ejemplo:** Hay 5 filósofos y 5 tenedores. Cada filósofo debe comer y pensar pero para comer necesita 2 tenedores, el que está a su derecha y el que está a su izquierda pero, cada tenedor puede ser tomado por un filósofo a la vez.
 - Cada tenedor es una sección crítica, podemos representar a los tenedores como un arreglo de semáforos.
 - Levantar un tenedor es hacer un **P**.
 - Bajar un tenedor es hacer un **V**.
 - Se genera deadlock si todos los filósofos hacen exactamente lo mismo, por ejemplo, todos primero levantan su tenedor derecho y luego el izquierdo, al querer levantar el izquierdo no pueden porque es el derecho que otro filósofo ya levantó.
- **Es un problema de exclusión mutua selectiva.**

Si en lugar de 5 filósofos fueran 3 ¿El problema sigue siendo de exclusión mutua selectiva?

- En el caso de que fueran 3 filósofos y no 5 como en la definición del problema general, no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos, dado que el resto de los procesos son sus adyacentes.

Si el problema es resuelto de forma centralizada y sin posiciones fijas ¿Siguiendo siendo de exclusión mutua selectiva?

- **Posiciones fijas:**
 - Se refiere a que cada filósofo solicita cubiertos y no necesariamente los de sus adyacentes.
- **Forma centralizada:**

- Los filósofos le consultan a un solo coordinador o mozo para poder tomar los cubiertos, en este caso, el mozo se los da si es que hay dos cubiertos disponibles sin tener en cuenta vecinos.
- Para este caso el problema deja de ser de exclusión mutua selectiva ya que compiten entre todos por acceder a los cubiertos.

Problema Lectores y Escritores - semáforos con exclusión mutua selectiva

- **Problema:** Dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los **escritores** debe ser exclusivo para evitar interferencia entre transacciones. Los **lectores** pueden ejecutar concurrentemente entre ellos si no hay **escritores** actualizando.
 - Los procesos son asimétricos y, según el scheduler, con diferente prioridad.
- **Es un problema de exclusión mutua selectiva.**

Solución como un problema de exclusión mutua

- Los **escritores** necesitan acceso exclusivo a la base de datos.
- Los **lectores** como grupo necesitan acceso exclusivo con respecto a los **escritores**.
- **Esta solución le da preferencia a los lectores → no es fair.**

Solución como un problema de exclusión mutua

- Cuando las **condiciones de espera** de los procesos son diferentes pero además son superpuestas, no es viable plantear una solución solo con el uso de semáforos, necesitamos usar la técnica **Passing the Baton**.

Si en el problema solo se acepta 1 escritor o un 1 lector en la base de datos ¿Tenemos un problema de exclusión mutua selectiva?

- Si se acepta solo 1 escritor y 1 lector el problema **deja de ser de exclusión mutua selectiva** ya que la competencia es contra todos.

Técnica **Passing the Baton**

- Técnica que **emplea un SBS para brindar exclusión** (en este caso a las variables que van a controlar el acceso a la base de datos) **y despertar procesos demorados siguiendo algún criterio.**
- **Puede usarse para implementar cualquier await.**
- **Cuando un proceso está dentro de la SC, mantiene el baton** que significa **permiso para ejecutar.**
- Cuando **el proceso llega a un SIGNAL (sale de la SC), pasa el baton a otro proceso.** Si ningún proceso está esperando por el **baton** (nadie quiere entrar a la SC) **el baton se libera para que lo tome el primer proceso que quiera entrar a la SC.**
- **Si ningún proceso está esperando una condición que sea true, el baton se pasa al próximo proceso que trata de entrar a su SC por primera vez.**
- Esta técnica **puede no cumplir la eventual entrada** ya que **el baton no se pasa a un proceso en particular**, sino que **es no determinístico**, por lo tanto, **puede quedar algún proceso colgado.** Se realiza un orden por **tipo de proceso** y no **por proceso específicamente.**
- La sincronización se expresa con sentencias atómicas de la forma:
 - **Incondicional** → $F_1: \langle S_i \rangle$
 - **Condicional** → $F_2: \langle \text{await } (B_j) S_j \rangle$
- **Componentes que necesita la técnica:**

- Un semáforo “e” inicialmente en 1 que controla la entrada a la SC.
- Un semáforo “b_j” inicialmente en 0, asociado con cada condición de espera.
- Por cada semáforo “b_j” un contador “d_j” inicializado en 0, que cuenta la cantidad de procesos dormidos en el semáforo “b_j” esperando que la condición se vuelva verdadera.
- El conjunto de los semáforos “e” y todos los “b_j” forman un SBS.
- **Traducción de las sentencias:**
 - **F₁:**
 - P(e);
 - S_i;
 - SIGNAL;
 - **F₂:**
 - P(e);
 - if (not B_i) {
 - d_j = d_j + 1;
 - V(e);
 - P(b_j);
 - }
 - S_j;
 - SIGNAL;
 - **SIGNAL:**
 - if (B₁ and d₁ > 0) {
 - d₁ = d₁ - 1;
 - V(b₁);
 - }
 - ...
 - elif (B_n and d_n > 0) {
 - d_n = d_n - 1;
 - V(b_n);
 - }
 - else V(e);

Técnica Passing the Condition

- Técnica utilizada **con Monitores** que consiste en **avisar al proceso que es despertado que están dadas las condiciones para que él continúe su ejecución** pero sin modificar el valor de la condición.

Relación y diferencias con Passing the Condition

- **Relación:**
 - Ambos **determinan cual es el proceso demorado que será despertado** para que ejecute su SC.
- **Diferencia:**
 - En Passing the Condition cuando un proceso hace signal pasa el control directamente al proceso que estaba esperando para poder entrar a la SC antes que otros procesos que llaman a un procedure dentro del monitor tengan chance de avanzar, a diferencia de la técnica de Passing the Baton que cuando un proceso hace signal es “globalmente visible” por todos los procesos que quieren entrar en la SC. **La técnica Passing the Condition pasa una condición directamente a un proceso despertado en lugar de hacerla globalmente visible.**

Alocación de Recursos y Scheduling

- **Problema:** Decidir cuándo se le puede **dar acceso** a un recurso a un proceso en específico. Este recurso puede ser un objeto, elemento, componente, SC, etc. que hace que el **proceso se demore al esperar adquirirlo**.
- **Definición del Problema:** Procesos que compiten por el uso de unidades de recurso **compartido** (cada unidad **está libre** o en **uso**)
 - El **proceso** tiene que hacer un **request(parámetros)** del/los recurso/s, **usar** el/los recurso/s y por último un **release(parámetros)** del/los recurso/s.
- **Podemos hacer uso de la técnica Passing the Baton**

solicitar
usar
liberar

```
request (parámetros): P(c);  
                        if (request no puede ser satisfecho) DELAY;  
                        tomar las unidades;  
                        SIGNAL;
```

```
release (parámetros): P(c);  
                      retornar unidades;  
                      SIGNAL;
```

Alocación Shortest-Job-Next (SJN)

- Esta política determina que **si hay más de un proceso esperando** por un recurso compartido, **lo usará el que lo tenga que usar por menos tiempo**.
- **Requerimiento del recurso:**
 - **request(tiempo_estimado_de_uso, id)**
 - Si el recurso está libre, es alocado inmediatamente al proceso **id**; sino, el proceso **id** se demora.
- **Release del recurso:**
 - **release()**
 - Cuando se libera el recurso, es alocado al proceso demorado (si lo hay) con el mínimo valor de **tiempo**. Si dos o más procesos tienen el mismo valor de **tiempo**, el recurso es alocado al que esperó más.
- Esta política minimiza el tiempo promedio de ejecución pero **es unfair**. **Se puede mejorar usando aging**.
- Si quisiéramos **respetar el orden de llegada**, deberíamos insertar los elementos en una cola y no haría falta enviar el tiempo de uso por parámetro en la request.
- Si tuviéramos **recursos de múltiples unidades**, podríamos llevar un contador de recursos libres y una estructura que guarde los recursos disponibles para ir sacando y reponiendo.

Monitores

- Módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.
- Es un **Mecanismo de abstracción** de datos que:
 - **Encapsula las representaciones de los recursos**.
 - **Brinda un conjunto de operaciones que son los únicos medios para manipular esos recursos**.
- **Contiene variables que almacenan el estado** del recurso y **procedimientos que implementan las operaciones sobre él**.
- **Actúan** de forma **pasiva**, es decir, estos únicamente **se ejecutan cuando un proceso activo de un programa concurrente invoca un procedure** de un **monitor**.

- La **comunicación y exclusión mutua** se resuelve de forma **implícita** ya que se asegura que los procedimientos que estén dentro de un mismo monitor no se ejecutan concurrentemente.
- La **sincronización por condición** se resuelve de forma **explícita** con el uso de variables condición.

Notación de los Monitores

- Los monitores se distinguen de un tipo de dato abstracto ya que los **monitores son compartidos por procesos que se ejecutan concurrentemente**. Tienen **interfaz y cuerpo**:
 - **Interfaz** → Especifica las operaciones (procedures) que brinda el recurso. Deja que solo los nombres de los procedimientos sean visibles desde afuera.
 - **Cuerpo** → Conjunto de variables que van a representar a ese recurso, se denominan **variables permanentes** (conservan su valor a través de las distintas invocaciones que se hacen a los procedimientos del monitor) e implementación de los procedimientos indicados en la interfaz del monitor.
 - Dentro del **Cuerpo**, los **procedimientos** sólo pueden acceder a **variables permanentes, sus variables locales y parámetros** que le sean pasados en la invocación.
- Los llamados al monitor tienen la forma:
 - **NombreMonitor.operacion_i(argumentos)**.
- El programador de un monitor **no puede conocer a priori el orden de llamado de los procedimientos del monitor**.

Implementación de la Sincronización por Condición

- Es programada explícitamente con **variables condición** → **cond cv**;
 - Son **variables permanentes del monitor que solo se pueden usar dentro del mismo**.
- Esta variable condición "cv" **tiene asociado internamente una cola de procesos dormidos/demorados que no es visible directamente** para el programador.
- **Operaciones permitidas** sobre las variables condición:
 - **wait(cv)** → El proceso que está ejecutando se demora al final de la cola de "cv" y **deja el acceso exclusivo al monitor**, es decir, lo libera para que otro proceso lo pueda usar.
 - **signal(cv)** → Despierta al proceso que está primero en la cola de procesos dormidos/demorados (si hay alguno) **y lo saca de la cola**. **El proceso despertado recién podrá ejecutar desde el punto en el que se durmió cuando readquiera el acceso exclusivo al monitor**.
 - **signal_all(cv)** → Despierta a todos los procesos dormidos/demorados en "cv", dejando vacía la cola asociada a esa variable.
- **Operaciones adicionales** sobre las variables condición que solo se usan en la teoría:
 - **empty(cv)** → Retorna **true** si la cola controlada por **cv** está vacía.
 - **wait(cv, rank)** → El proceso se demora en la cola de **cv** en orden ascendente de acuerdo al parámetro **rank** y deja el acceso exclusivo al monitor.
 - **minrank(cv)** → Función que retorna el **mínimo ranking de demora**, es decir, retorna el ranking del primer proceso demorado en la cola.

Protocolos de Sincronización - Disciplinas de Sincronización

- Indican cuándo el proceso que es despertado va a poder volver a acceder al monitor de forma exclusiva para continuar su ejecución.

Signal and Continue

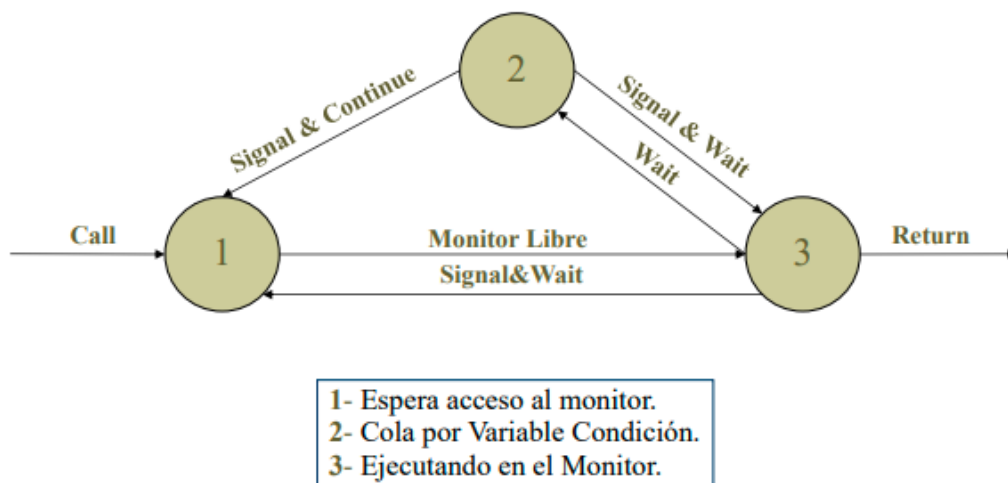
- El proceso que hace el **signal(cv)** despierta al proceso que estaba dormido pero sigue haciendo uso del monitor hasta que termina de hacer uso del mismo o es dormido.
- **El proceso que fue despertado tendrá que competir para volver a acceder al monitor.**

Signal and Wait

- El proceso que hace el **signal(cv)** pasa a ser quien tendrá que competir nuevamente por el acceso exclusivo al monitor ya que **el proceso despertado es quien toma control del mismo.**

Diferencias

- **Signal and Continue**
 - El proceso que hace el signal continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).
- **Signal and wait**
 - El proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.



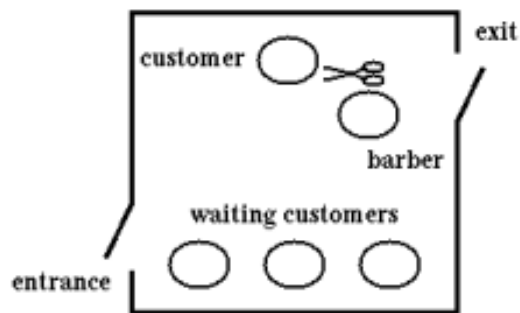
Problemas Concurrentes y Técnicas con Monitores

Alocación SJN - Wait con Prioridad y Variables Condición Privadas

- **Tenemos 2 soluciones posibles:**
 - **Wait con Prioridad:**
 - Hacemos uso de la operación wait con prioridad que se puede usar en la teoría y lo combinamos con Passing the Condition.
 - **Variables Condición Privadas:**
 - Se usa Passing the Condition, manejando el orden explícitamente por medio de una cola ordenada y variables condición privadas.

Problema del Peluquero Dormilón - Rendezvous

- **Descripción:**
 - Una ciudad tiene una peluquería con 2 puertas y unas pocas sillas. Los clientes entran por una puerta y salen por la otra. Como el negocio es chico, **a lo sumo un cliente o el peluquero se pueden mover en él a la vez**. El peluquero pasa su tiempo atendiendo clientes, **uno por vez**. Cuando no hay ninguno, **el peluquero duerme en su silla**. Cuando llega un cliente y encuentra que el peluquero está durmiendo, **el cliente lo despierta, se sienta en la silla del peluquero, y duerme mientras el peluquero le corta el pelo**. Si el peluquero está ocupado cuando llega un cliente, **éste se va a dormir en una de las otras sillas**. Después de un corte de pelo, el peluquero **abre la puerta de salida para el cliente y la cierra cuando el cliente se va**. Si hay clientes esperando, **el peluquero despierta a uno y espera que se siente**. Sino, **se vuelve a dormir hasta que llegue un cliente**.
- **Tipos del proceso del problema** → clientes y peluquero
- **Monitor** → administrador de la peluquería con tres procedures:
 - **corte_de_pelo** → llamado por los clientes, que retornan luego de recibir un corte de pelo.
 - **proximo_cliente** → llamado por el peluquero para esperar que un cliente se siente en su silla, y luego le corta el pelo.
 - **corte_terminado** → llamado por el peluquero para que el cliente deje la peluquería.
- **Etapas de sincronización entre un cliente y el peluquero (rendezvous):**
 1. El peluquero tiene que esperar a que llegue un cliente, y este tiene que esperar que el peluquero esté disponible.
 2. El cliente necesita esperar que el peluquero termine de cortar el pelo para poder irse.
 3. Antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente se haya ido.



Problema de Scheduling de Disco

- El disco contiene "platos" conectados a un eje central y que rotan a velocidad constante. Las pistas forman círculos concéntricos → concepto de cilindro de información.
- Los datos se acceden posicionando una cabeza lectora/escritora sobre la pista apropiada, y luego esperando que el plato rote hasta que el dato pase por la cabeza.
 - **dirección física** → cilindro, número de pista, y desplazamiento
- **Para acceder al disco, un programa ejecuta una instrucción de E/S con los parámetros:**
 - Dirección física del disco.
 - Número de bytes a transferir.
 - Tipo de transferencia (Read o Write).
 - Dirección de un buffer.

- **El tiempo de acceso al disco depende de:**
 - A. Seek time para mover una cabeza al cilindro apropiado.
 - B. Rotational delay.
 - C. Transmission time (**depende solo del número de bytes**).
- **A. y B. dependen del estado del disco** (en qué cilindro, en qué sector está y hacia dónde tengo que ir) → buscamos minimizar el **tiempo de seek**.
- **Políticas de Scheduling de un disco:**
 - **Shortest-Seek-Time (SST).**
 - **SCAN, LOOK, o algoritmo del ascensor.**
 - **CSCAN o CLOOK**
 - Se atienden pedidos en una sola dirección. Es fair y reduce la varianza del tiempo de espera.
 - Este es el que nos interesa.

Solución con Monitor Separado

- El **scheduler** es implementado por un monitor para que los datos sean accedidos solo por un proceso usuario a la vez. El monitor provee dos operaciones: **pedir y liberar**.
 - **Scheduler_Disco.pedir(cil) - Accede al disco - Scheduler_Disco.liberar()**
- Suponemos cilindros numerados de **0 a MAXCIL** y **scheduling CSCAN**.
- A lo sumo **un proceso a la vez puede tener permiso para usar el disco, y los pedidos pendientes son servidos en orden CSCAN**.
 - Hay que **distinguir entre los pedidos pendientes a ser servidos en el scan corriente y los que serán servidos en el próximo scan**.
- **posición** es la variable que indica posición corriente de la cabeza (en qué cilindro está parada la cabeza).
- **Problemas de esta solución:**
 - La presencia del scheduler es visible al proceso que usa el disco. Si se borra el scheduler, los procesos usuario cambian.
 - Todos los procesos usuario tienen que seguir el protocolo de acceso, de no ser así el scheduling falla.
 - Luego de obtener el acceso, el proceso debe comunicarse con el driver de acceso al disco a través de 2 instancias de buffer limitado.

Solución con Monitor Intermedio

- Para mejorar la solución anterior usamos un **monitor intermedio** entre los procesos usuario y **1 monitor** el disk driver. **El monitor envía los pedidos al disk driver en el orden de preferencia deseado.**
 - **Mejoras:**
 - **La interfaz al disco usa un único monitor, y los usuarios hacen un solo llamado al monitor por acceso al disco.**
 - La existencia o no de scheduling es transparente.
 - No hay un protocolo multipaso que deba seguir el usuario y en el cual pueda fallar.
- clts dejan pedido al monitor y el proceso disk driver resuelve, con un orden

Conceptos Generales del Procesamiento Distribuido

- Los **procesos solo comparten canales (físicos o lógicos)** que pueden ser de 3 variantes:
 - **Mailbox, input port, link.**
 - **Uni o Bidireccionales.**
 - **Sincrónicos o Asincrónicos.**
- Existen diversos **mecanismos para el Procesamiento Distribuido:**

- Pasaje de Mensajes Asíncronos (PMA).
- Pasaje de Mensajes Sincrónicos (PMS).
- Llamado a Procedimientos Remotos (RPC).
- Rendezvous.

Dualidad entre los mecanismos de pasaje de mensajes y monitores

- Se puede considerar que existe una dualidad entre los mecanismos de monitores y pasaje de mensajes porque cada uno puede simular al otro. Esto significa que cualquier problema de sincronización que se pueda resolver con monitores también se puede resolver con pasaje de mensajes, y viceversa.

<i>Programas con Monitores</i>		<i>Programas basados en PM</i>
• Variables permanentes	↔	Variables locales del servidor
• Identificadores de procedures	↔	Canal <i>request</i> y tipos de operación
• Llamado a procedure	↔	<i>send request()</i> ; <i>receive respuesta</i>
• Entry del monitor	↔	<i>receive request()</i>
• Retorno del procedure	↔	<i>send respuesta()</i>
• Sentencia <i>wait</i>	↔	Salvar pedido pendiente
• Sentencia <i>signal</i>	↔	Recuperar/ procesar pedido pendiente
• Cuerpos de los procedure	↔	Sentencias del “case” de acuerdo a la clase de operación.

Características de los Canales

- El acceso a los contenidos de cada canal se hace de forma atómica y respetando el orden FIFO.
- En principio son ilimitados, aunque en una implementación real se ven limitados por un tamaño de buffer.
- Los mensajes no se pierden ni se modifican y todo mensaje enviado en algún momento puede ser “leído”.
- La función `empty(ch)` nos sirve para determinar si la cola de un canal está vacía.

Canal Mailbox

- En estos canales cualquier proceso puede enviar o recibir por alguno de los canales declarados.
- Pueden haber varios emisores y varios receptores en el mismo canal.

Canal Input port

- En estos canales solo puede haber un receptor y pueden haber varios emisores.

Canal Link

- En estos canales solo puede haber un receptor y un emisor.

Relación entre mecanismos de sincronización

- Semáforos mejora respecto del busy waiting.

- Los **monitores** combinan Exclusión Mutua implícita y señalización explícita.
- **Pasaje de Mensajes** extiende semáforos con datos.
- **RPC y Rendezvous** combinan la interfaz procedural de monitores con PM implícito.

Pasaje de Mensajes Asincrónicos

- Mecanismo de Sincronización para procesamiento distribuido.
- **Operación Send:**
 - Un **proceso** agrega un mensaje al final de la cola “ilimitada” de un canal ejecutando un send, que **no bloquea** al emisor.
- **Operación Receive:**
 - Un **proceso** recibe un mensaje desde un canal con receive, que demora “bloquea” al **receptor** hasta que en el canal haya al menos un mensaje; luego **toma el primero y lo almacena en variables locales**.
 - Es una primitiva **bloqueante** ya que produce un delay.
 - **Semánticamente**, el **proceso no hace nada hasta recibir un mensaje** en la cola correspondiente al canal. **No es necesario hacer polling.**

Problemas Concurrentes y Técnicas con PMA

Productores y Consumidores - Filtros

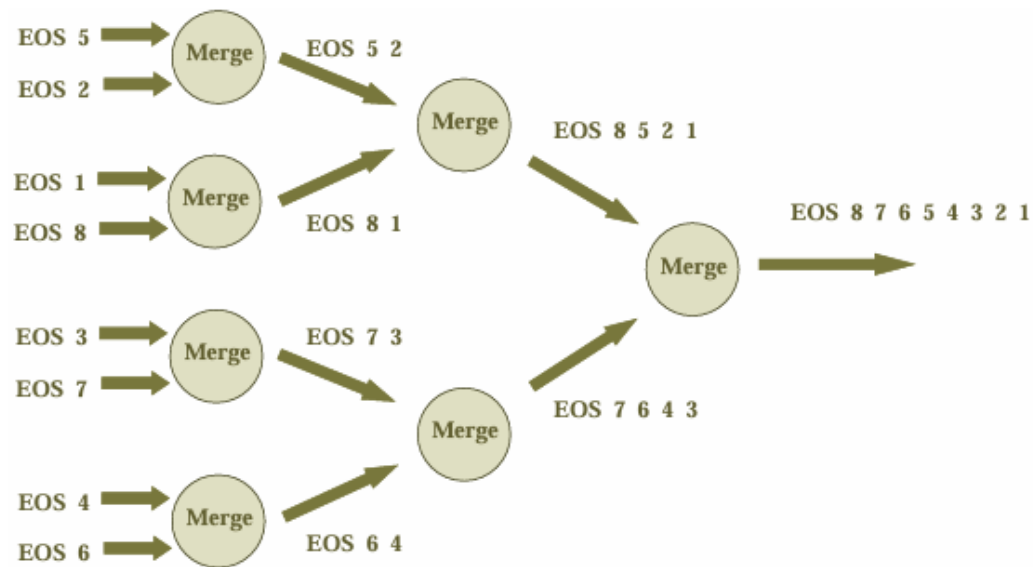
- **Problema:**
 - Ordenar una lista de N números de modo ascendente. Podemos pensar en una solución “secuencial” que usa un filtro **Sort** con un canal de entrada (N números desordenados) y un canal de salida (N números ordenados).
 - **Sort** puede determinar que recibió todos los números de estas 3 formas:
 - Conoce N.
 - Envía N como el primer elemento a recibir por el canal de entrada.
 - Cierra la lista de N números con un valor especial o “centinela”.

Filtros

- Proceso que **recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida**.
- La **salida de un filtro es función de su estado inicial y de los valores recibidos**.
- Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada.

Solución con Red de Ordenación

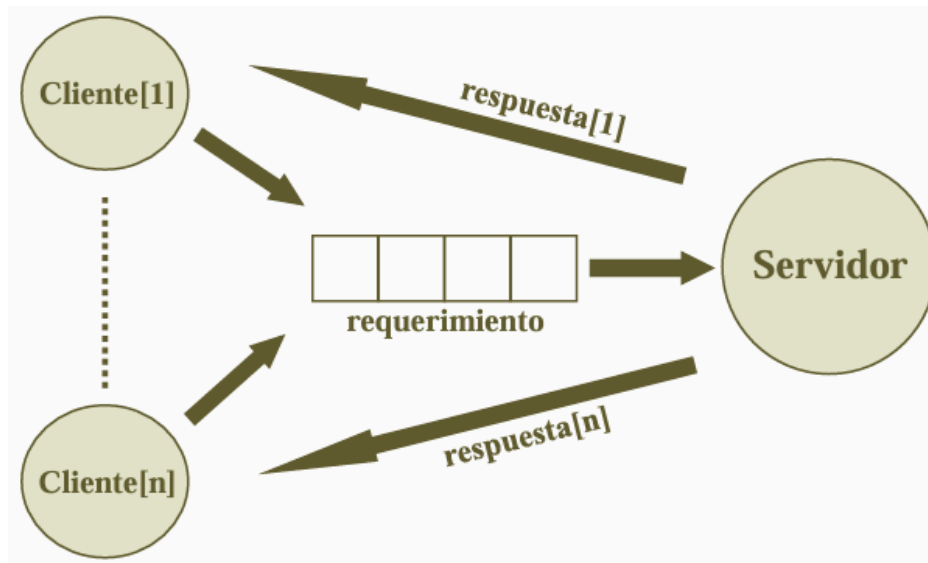
- Solución más eficiente que la secuencial.
- Usamos una **red de pequeños procesos que ejecutan en paralelo** e interactúan para “armar” la salida ordenada (**merge network**).
- La idea es **mezclar repetidamente y en paralelo dos listas ordenadas** de “Y” elementos cada una en una lista ordenada de $2 * Y$ elementos.
 - Con PMA pensamos en 2 canales de entrada de datos, uno de salida y usamos el centinela para marcar el fin de las secuencias.



- Para saber la **cantidad de procesos de la red** hacemos $N - 1$ siendo N la cantidad de números a ordenar.
 - Para saber el **ancho de la red** hacemos $\log_2(N)$ siendo N la cantidad de números a ordenar.
 - Puede programarse usando:
 - **Static Naming:**
 - Arreglo global de canales, y cada instancia de Merge recibe 2 elementos del arreglo y envía otro.
 - **Dynamic Naming:**
 - Canales globales, parametrizar los procesos, y darle a cada proceso 3 canales al crearlo; todos los Merge son idénticos pero se necesita un coordinador.
- Coordinador da 3 canales, 2 de entrada y 1 de salida.

Clientes y Servidores

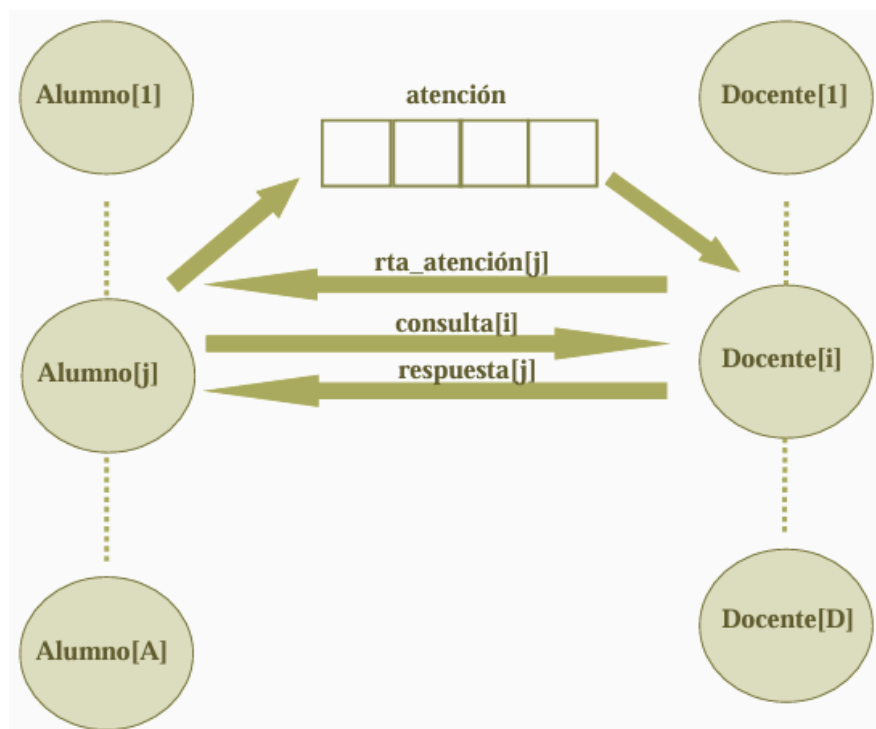
- **Servidor:**
 - Proceso que maneja pedidos de otros procesos clientes.
- **Cliente:**
 - Procesos que envían mensajes a un canal de requerimientos general, luego reciben los resultados desde un canal de respuesta propio.



Comunicación para 1 operación

Continuidad Conversacional

- Un proceso cliente hace un requerimiento a un proceso servidor y **una vez que el servidor lo atiende el proceso cliente comienza a realizar varios requerimientos hasta que no le quedan requerimientos para hacer, recién en este punto el servidor vuelve a quedar libre.**



Comunicación para varias operaciones

Pares que interactúan

- **Problema:**
 - Cada proceso tiene un dato local V y los N procesos deben saber cuál es el menor y cuál el mayor de los valores.

Solución Simétrica

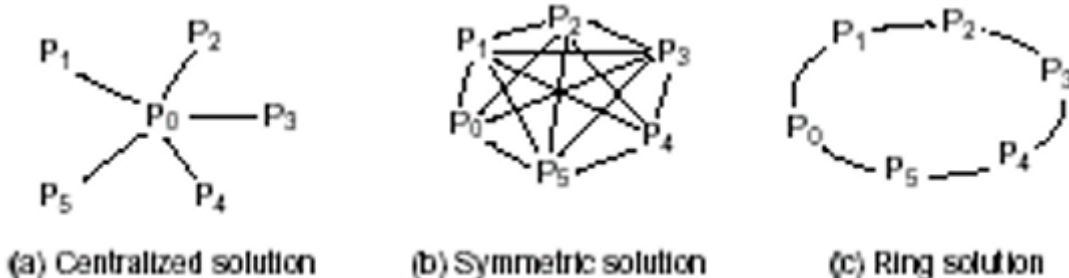
- Es la solución más corta y sencilla de programar.
- Hay un canal entre cada par de procesos.
- Usa el mayor número de mensajes si es que no hay broadcast:
 - Sin broadcast son $N * (N - 1)$ mensajes siendo N la cantidad de procesos.
 - Con broadcast son N mensajes.

Solución Centralizada

- Se envían mensajes a un coordinador casi al mismo tiempo, solo el primer receive del coordinador demora mucho.
- Usa un número lineal de mensajes.
 - $2 * (N - 1)$ mensajes.

Solución Anillo

- En esta solución todos los procesos son productores y consumidores. El último tiene que esperar a que todos los demás (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado.
- Los mensajes circulan 2 veces completas por el anillo.
- Es una solución lineal y lenta.
- Usa un número lineal de mensajes.
 - $(2 * N) - 1$ mensajes.



Pasaje de Mensajes Sincrónicos

- Mecanismo de Sincronización para procesamiento distribuido.
- En PMS tanto el **send** como el **receive** son primitivas bloqueantes.
 - Si un proceso trata de enviar a un canal se demora hasta que otro proceso está esperando recibir por ese canal. De esta manera, un emisor y un receptor se sincronizan en todo punto de comunicación.
- La cola de mensajes asociada a un send sobre un canal se reduce a un mensaje. Esto significa menor memoria.
- Los procesos no comparten los canales.
- El grado de concurrencia se reduce respecto a PMA.
- Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar.
- Con PMS se tiene mayor probabilidad de deadlock.
- El programador debe ser cuidadoso para que todos los send y receive hagan matching. Es ideal para algoritmos del tipo Cliente/Servidor o de Filtros.
- Los canales son de tipo link.

CSP - Lenguaje para Pasaje de Mensajes Sincrónicos

- **Canal:**
 - Link directo entre dos procesos en lugar de mailbox global. Son halfduplex y nominados.
- Las sentencias de Entrada (? o query) y Salida (! o shriek o bang) son el único medio por el cual los procesos se comunican.
- Para que se produzca la comunicación, deben matchear, y luego se ejecutan simultáneamente.
- Efecto: sentencia de asignación distribuida.

Comunicación guardada

- Las operaciones de comunicación (? y !) pueden ser guardadas, es decir hacer un AWAIT hasta que una condición sea verdadera → El do e if de CSP usan los comandos guardados de Dijkstra.
- Permite a los procesos comunicarse de forma no determinística y selectiva. En lugar de bloquearse en una operación de entrada o salida, un proceso puede esperar a que se cumplan ciertas condiciones o a que se reciban mensajes específicos.
- **Brinda:**
 - Flexibilidad en la comunicación.
 - Prevención de bloqueos.
 - Manejo eficiente de la concurrencia.
- Tienen la forma $B; C \rightarrow S;$

Las sentencias de comunicación guardadas aparecen en *if* y *do*.

$$\begin{array}{l} \text{if } B_1; \text{comunicación}_1 \rightarrow S_1; \\ \bullet \quad B_2; \text{comunicación}_2 \rightarrow S_2; \\ \text{fi} \end{array}$$

Resultados de la evaluación de una guarda

- **Éxito:**
 - La condición booleana **B** es verdadera (o no existe) y la comunicación **C** se puede realizar sin demora.
- **Fallo:**
 - La condición booleana **B** es falsa, independientemente de la sentencia de comunicación.
- **Bloqueo:**
 - La condición booleana **B** es verdadera (o no existe), pero la comunicación **C** no se puede realizar sin demora.

Ejecución

1. Primero, se evalúan las guardas:
 - a. Si todas las guardas fallan, el if termina sin efecto.
 - b. Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
 - c. Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.

2. Segundo, luego de elegir una guarda exitosa, **se ejecuta la sentencia de comunicación de la guarda elegida.**
3. Tercero, **se ejecuta la sentencia S.**
- **La ejecución del "do" es similar (se repite hasta que todas las guardas fallen).**

Ejemplo - Criba de Eratóstenes 🧠

- **Problema:**
 - **Generar todos los números primos entre 2 y N \rightarrow 2, 3, 4, 5, 6, 7, 8, ..., N.**
- **Comenzando con el primer número (2), recorreremos la lista y borramos los múltiplos de ese número. Si N es impar: 2, 3, 5, 7, ..., N.**
- **Pasamos al próximo número (3) y borramos sus múltiplos.**
- **Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y N.**
- **La criba captura primos y deja caer múltiplos de los primos.**

```

1  ProcessCriba[1] {
2      int p = 2;
3      for (i = 3 to n by 2) do
4          Criba[2] ! (i);
5      od
6      Criba[2] ! (-1);
7  }
8
9  ProcessCriba[i = 2 to L] {
10     int p, proximo;
11     bool terminar = false;
12     Criba[i-1] ? (p);
13     if (p = -1) →
14         if (i < L) →
15             Criba[i+1] ! (-1);
16         fi
17     else →
18         print("Primo: ", p);
19         do (!terminar) Criba[i-1] ? (proximo) →
20             if (proximo = -1) →
21                 if (i < L) →
22                     Criba[i+1] ! (-1);
23                 fi
24                 terminar = true;
25             fi
26             if ((proximo MOD p) <> 0) and (i < L) →
27                 Criba[i+1] ! (proximo);
28             fi
29         od
30     fi
31 }

```

Annotations:

- Line 2: `int p = 2;` → `print(p);` (imprimo 2, paso todos los q no son múltiplos 2 al siguiente (hasta -1 que corta ejecución))
- Line 15: `Criba[i+1] ! (-1);` → Si recibo -1 no hay más números, si no soy el ultimo envío -1
- Line 18: `print("Primo: ", p);` → Imprimo mi numero (primo)
- Line 22: `Criba[i+1] ! (-1);` → Recibo números y si no son múltiplos de mi primo se lo envío al siguiente, si tengo q terminar y no soy el ultimo le envío -1 al siguiente

Criba sin Deadlock

Paradigmas de Interacción entre Procesos

- **3 esquemas básicos de interacción entre procesos:**
 - productor/consumidor, cliente/servidor e interacción entre pares.
- Estos esquemas básicos **se pueden combinar** de muchas maneras, dando lugar a otros paradigmas o modelos de interacción entre procesos.

Paradigma Master/Worker

- El concepto de **bag of tasks** usando variables compartidas supone que un conjunto de workers comparten una “bolsa” con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa.
- Este paradigma **es la implementación distribuida del modelo de bag of tasks que consiste en un proceso controlador de datos y/o procesos y múltiples procesadores que acceden a él para poder obtener datos y/o tareas para ejecutarlos en forma distribuida.**

Paradigma Heartbeat

- Los procesos periódicamente **deben intercambiar información y para hacerlo ejecutan dos etapas**; en la primera se expande enviando información (**SEND a todos**) y en la segunda se contrae adquieren información (**RECEIVE de todos**).
- **Su uso** más importante **es paralelizar soluciones iterativas.**

Topología de una red

- Los **procesadores están conectados por canales bidireccionales**. Cada uno se comunica sólo con sus vecinos y conoce esos links.
- **Excesivo intercambio de mensajes** los procesos cercanos al “centro” conocen la topología más pronto y no aprenden nada nuevo en los intercambios.

Paradigma Pipeline

- **Pipeline:**
 - **Arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.**
- La información recorre una serie de procesos utilizando alguna forma de receive/send donde la salida de un proceso es la entrada del siguiente.
- **Esquemas posibles:**
 - **Esquema a lazo abierto:**
 - **W_1 en el INPUT, W_n en el OUTPUT.**
 - **Esquema de pipeline circular:**
 - **W_n se conecta con el W_1 .**
 - **Esquema cerrado:**
 - **Un proceso coordinador maneja la realimentación entre W_n y W_1 .**

Paradigma Probe/Echo

- Se basa en el **envío de un mensaje (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”).**
- Los **probes se envían en paralelo a todos los sucesores.**

Paradigma Broadcast

- Permiten **alcanzar una información global en una arquitectura distribuida**. Sirven para la toma de decisiones descentralizadas y para resolver problemas de sincronización distribuida.
- En este paradigma, **un proceso envía un mensaje idéntico a todos los demás procesos** en el sistema.
- **Broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.**

Paradigma Token Passing

- Un paradigma de interacción muy usado se basa en un **tipo especial de mensaje ("token") que puede usarse para otorgar un permiso (control) o recoger información global** de la arquitectura distribuida.

Paradigma Servidores Replicados

- Un server puede ser replicado cuando **hay múltiples instancias de un recurso: cada server maneja una instancia**.
- La replicación también **puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios**.
- Ejemplo de los Filósofos:
 - **Modelo Centralizado:**
 - Los filósofos se comunican con 1 mozo que decide el acceso a los recursos.
 - **Modelo Distribuido:**
 - Se supone 5 mozos cada uno manejando un tenedor. Un filósofo puede comunicarse con 2 mozos (el de la izquierda y el de la derecha), solicitando y devolviendo el recurso.
 - Los mozos no se comunican entre ellos.
 - **Modelo Descentralizado:**
 - Cada filósofo ve a un único mozo.
 - Los mozos se comunican entre ellos (cada uno con el de su derecha y el de su izquierda) para decidir el manejo del recurso asociado a "su" filósofo.

RPC y Rendezvous

- Tanto **RPC (Llamada a Procedimiento Remoto)** como **Rendezvous** son mecanismos de **comunicación y sincronización** entre procesos **en sistemas distribuidos**, especialmente adecuados para aplicaciones cliente/servidor. Ambos **combinan una interfaz similar a la de los monitores**, con operaciones exportadas invocables mediante llamadas externas (CALL), con el **uso de mensajes sincrónicos**. Esto significa que el **proceso que realiza la llamada se bloquea hasta que la operación llamada se complete y se devuelvan los resultados**.
 - El **Pasaje de Mensajes** se ajusta bien a problemas de **filtros y pares que interactúan**, ya que se plantea la **comunicación unidireccional**.
 - **RPC y Rendezvous** se ajustan bien a problemas de **cliente y servidor** ya que usan **técnicas de comunicación y sincronización** que **suponen un canal bidireccional**, lo que es ideal para ese tipo de problemas.

Similitudes

- **Interfaz similar a un monitor:** Ambos mecanismos permiten definir módulos con operaciones (o procedimientos) exportadas que pueden ser invocadas por procesos en otros módulos.
- **Comunicación sincrónica:** En ambos casos, la llamada a una operación es bloqueante. El proceso llamador se suspende hasta que el proceso servidor complete la operación y devuelva los resultados.
- **Orientación a Cliente/Servidor:** RPC y Rendezvous son especialmente útiles para implementar interacciones cliente/servidor, donde un servidor ofrece servicios a múltiples clientes.

Diferencias

- **Manejo de la invocación:** La principal diferencia radica en cómo se maneja la invocación de una operación:
 - **RPC:** Se crea un nuevo proceso (o se utiliza uno de un pool preexistente) para manejar cada llamada. El proceso servidor ejecuta el procedimiento correspondiente a la operación, envía los resultados al llamador y finaliza.
 - **Rendezvous:** La Invocación se realiza mediante un "encuentro" (rendezvous) con un proceso servidor ya existente. Este proceso utiliza una sentencia de entrada (in) para esperar una invocación, procesarla y devolver los resultados. La atención de las operaciones se realiza de forma secuencial, no concurrente.
- **Sincronización:**
 - **RPC:** El mecanismo en sí mismo sólo proporciona comunicación. La sincronización entre los procesos dentro de un módulo debe programarse explícitamente utilizando semáforos, monitores u otros mecanismos.
 - **Rendezvous:** Combina comunicación y sincronización. La sentencia de entrada (in) del proceso servidor se encarga de la sincronización, bloqueando al servidor hasta que llegue una invocación y al cliente hasta que se complete la operación.

RPC (Remote Procedure Call)

- Los programas se descomponen en módulos (con procesos y procedures), que pueden residir en espacios de direcciones distintos.
- Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo.
- Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
- Los procesos locales son llamados background para distinguirlos de las operaciones exportadas.
- Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:
 - **call Meneame.opname(argumentos).**
 - Para un llamado local el nombre del módulo se puede omitir.
- La implementación de un llamado intermodulo es distinta que para uno local, ya que los dos módulos pueden estar en distintos espacios:
 - Un nuevo proceso sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server.
- Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.
- En general, un llamado será remoto se debe crear un proceso server o alocarlo de un pool preexistente.

- Es necesario proveer sincronización dentro de los módulos porque los procesos pueden ejecutarse concurrentemente. Estos mecanismos de sincronización pueden usarse tanto para el acceso a variables compartidas como para sincronizar interacciones entre los procesos si fuera necesario.

Enfoques para proveer sincronización

- Si los procesos se ejecutan por **exclusión mutua**, sólo hay un proceso activo a la vez dentro del módulo, el acceso a las variables compartidas tiene exclusión mutua implícita pero la sincronización por condición debe programarse. Pueden usarse sentencias await o variables condición.
- Si los procesos se ejecutan **concurrentemente** dentro del módulo debe implementarse tanto la exclusión mutua como la sincronización por condición para esto pueden usarse cualquiera de los mecanismos existentes como semáforos, monitores, PM o Rendezvous.

Rendezvous

- Combina comunicación y sincronización.
- Como con RPC, un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo.
- Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar.
- Las operaciones se atienden una por vez más que concurrentemente.
- A diferencia de RPC el servidor es un proceso activo.

¿Qué elemento de la forma general de Rendezvous no se encuentra en ADA?

- Rendezvous provee la posibilidad de asociar sentencias de scheduling y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling. Ada no provee estas posibilidades.

ADA - Lenguaje con Rendezvous

- Desde el punto de vista de la concurrencia, un programa Ada tiene tasks (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
- Los puntos de invocación (entrada) a una tarea se denominan **entrys** y están especificados en la parte visible (header de la tarea).
- Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva **accept**.
- La tarea que declara un entry sirve llamados al entry con **accept**:
 - **accept nombre (parámetros formales) do sentencias end nombre;**
- Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

Comunicación Guardada

- La sentencia "select" permite al servidor elegir entre diferentes alternativas de **accept**.
- Cada línea del select es una "alternativa" que puede incluir una cláusula "when" para especificar una condición.
- Se puede usar "else", "delay" o "terminate" en las alternativas.

Thread

- Proceso “liviano” que tiene su propio contador de programa y su pila de ejecución, pero no controla el “contexto pesado” (por ejemplo, las tablas de página).

POSIX Threads - Pthreads

- Modelo de ejecución que existe independientemente del lenguaje, así como un modelo de ejecución paralela. Permite a un programa controlar múltiples flujos de trabajo diferentes que se solapan en el tiempo.
- Cada flujo de trabajo se denomina hilo (o thread), y la creación y el control de estos flujos se logra mediante llamadas a la API de POSIX Threads.
- El “main” termina cuando terminan todos los hilos.

Funciones Básicas de Pthreads

- **int pthread_create(pthread_t *thread_handle, const pthread_attr_t *attribute, void* (*thread_function)(void*), void *arg);**
 - Sirve para crear un hilo.
 - **Argumentos de la función:**
 - Primero el manejador, cuando se quiera referir al hilo se refiere al manejador.
 - El segundo es un puntero a una estructura de atributos del hilo. Esta estructura permite especificar configuraciones del hilo, como el tamaño de la pila, la prioridad, el nivel de concurrencia, etc.
 - El tercero es la función que se ejecutará en el nuevo hilo.
 - El cuarto es un puntero a los datos que se pasarán como argumento a la función del hilo. Esto permite que la función reciba parámetros personalizados. conviene enviar constantes (buena practica)
- **int pthread_exit (void *res);**
 - Se usa para terminar la ejecución de un hilo de manera explícita y devolver un valor de salida a quien lo espere.
 - Debe estar al final de la función que se pasa como tercer parámetro en el create.
- **int pthread_join (pthread_t thread, void **ptr);**
 - Se usa para esperar a que un hilo termine su ejecución. Es una función de sincronización que permite que el hilo que la llama espere hasta que el hilo especificado haya finalizado. **int pthread_join (pthread_t thread, void **retval);**
- **int pthread_cancel (pthread_t thread);**
 - Se usa para cancelar (terminar) un hilo en ejecución en programas multihilo.
 - Esta función envía una solicitud de cancelación al hilo especificado, pero la terminación del hilo no es inmediata, ya que depende del estado de cancelación del hilo y de si permite recibir solicitudes de cancelación.

Sincronización por Exclusión Mutua

- La librería posee la herramienta de mutex locks (**pthread_mutex**) para manejar la sincronización por exclusión mutua, **pthread_mutex** es una variable que tiene 2 estados posibles (bloqueado y desbloqueado), y solo un thread puede bloquear el mutex, ya que la operación Lock es atómica. Para poder entrar a una sección crítica un thread debe haber bloqueado el mutex.

Funciones

- **int pthread_mutex_lock (pthread_mutex_t *mutex);**
 - Bloquear.
- **int pthread_mutex_unlock (pthread_mutex_t *mutex);**
 - Desbloquear.
- **int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *lock_attr);**
 - Se usa para inicializar un mutex.
 - Argumentos:
 - El **primero** es un puntero a una variable del tipo **pthread_mutex_t** que se va a **inicializar como un mutex**.
 - El **segundo** es un puntero a una **estructura de atributos del mutex**. Se puede **usar para configurar el comportamiento del mutex** (por ejemplo, si es recursivo, el protocolo de bloqueo, etc.).
 - **Tipos** de mutex:
 - **Normal:**
 - Un Mutex con el atributo Normal **NO** permite que un **thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock)**.
 - **Recursive:**
 - Un Mutex con el atributo Normal **SI** permite que un **thread que lo tiene bloqueado vuelva a hacer un lock sobre él (deadlock)**.
 - **Error Check:**
 - Trabaja como el normal, **solo que en lugar de dejar bloqueado al proceso, se informa error**.
 - Para evitar tiempos ociosos se puede utilizar la función **pthread_mutex_trylock**. Que retorna el control informando si pudo hacer o no el lock.
 - Es **no bloqueante**:
 - Si un mutex está desbloqueado, lo bloquea e informa que se puede realizar el bloqueo; si un mutex está bloqueado, informa que no se puede hacer el bloqueo y retorna el control.

Sincronización por Condición

- Para manejar la sincronización por condición utiliza como herramienta las variables condición (**pthread_cond**), estas están **asociadas a un predicado que cuando se convierte en true da señal a todos los threads que están esperando** en la misma.
- Cada variable condición siempre **tiene un mutex asociado, el cual es bloqueado por cada thread para testear el predicado** definido en la variable.
- Si el predicado es falso, el thread **espera en la variable condición utilizando la función pthread_cond_wait**.

Funciones

- **int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)**
 - Se usa para que un hilo **espere una condición mientras mantiene la sincronización con un mutex**. Se usa para evitar esperas activas y garantizar que un hilo solo continúe su ejecución cuando una determinada condición sea señalada por otro hilo.
 - El hilo se duerme si la **variable es falsa, continua si es verdadera**.

- **int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex const struct timespec *abstime)**
 - Se usa para que un hilo espere una condición durante un tiempo limitado mientras mantiene la sincronización con un mutex. Si la condición no se cumple dentro del tiempo especificado, la función termina con un error indicando que el tiempo expiró.
- **int pthread_cond_signal (pthread_cond_t *cond)**
 - Se usa para despertar un solo hilo que está esperando en una variable de condición.
- **int pthread_cond_broadcast (pthread_cond_t *cond)**
 - Se usa para despertar a todos los hilos que están esperando en una variable de condición.
- **int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr)**
 - Se usa para inicializar una variable de condición.
 - Argumentos:
 - El primero es un puntero a la variable de condición que se va a inicializar.
 - El segundo es un puntero a una estructura de atributos (pthread_condattr_t). Permite establecer atributos específicos para la variable de condición, como si se compartirá entre procesos o no.
- **int pthread_cond_destroy (pthread_cond_t *cond)**
 - Se usa para destruir una variable de condición previamente inicializada con pthread_cond_init. Esta función libera los recursos asignados a la variable de condición.

Attribute Object

- API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
- Es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición).
- Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
- **Ventajas:**
 - Mejora la modularidad.
 - Facilidad de modificación del código.

Semáforos

- Los threads se pueden sincronizar por semáforos (librería semaphore.h).
- **Algunas Funciones:**
 - **sem_t semáforo**
 - Se declaran globales a los threads.
 - **sem_init (&semáforo, alcance, inicial)**
 - En esta operación se inicializa el semáforo. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos (≠ 0).
 - **sem_wait(&semáforo)**
 - Equivale al P.
 - **sem_post(&semáforo)**
 - Equivale al V.

Monitores

- No se posee Monitores pero con la **exclusión mutua** y la **sincronización por condición** se pueden simular.
 - El **acceso exclusivo al monitor** se simula usando una **variable mutex** la cual se bloquea antes de la llamada al procedure y se desbloquea al terminar el mismo (una variable mutex diferente para cada monitor).
 - **Cada llamado de un proceso a un procedure de un monitor debe ser reemplazado por el código de ese procedure.**

Librerías para manejo de Pasaje de Mensajes

- **Hay diferentes protocolos para SEND y RECEIVE:**
 - **Send bloqueante:** El transmisor queda esperando que el mensaje sea recibido por el receptor (PMS).
 - **Send no bloqueante:** El transmisor no queda esperando que el mensaje sea recibido por el receptor (PMA).
 - **Send con buffering:** La cola de mensajes asociada con el canal del send puede acumular mensajes (PMA).
 - **Send sin buffering:** La cola de mensajes asociada con el canal del send se reduce a 1 (PMS).
- Esos mismos conceptos de **bloqueante, no bloqueante, con buffering y sin buffering** se aplican al **RECEIVE**.

MPI - Message Passing Interface

- **Biblioteca de comunicaciones** a través de pasaje de mensajes que **permite comunicar y sincronizar procesos secuenciales escritos en diferentes lenguajes** que se ejecutan sobre una arquitectura distribuida.
- **El estilo de programación es SPMD.**
 - En informática, SPMD (**programa único, datos múltiples**) es una técnica empleada para lograr el paralelismo; **es una subcategoría de MIMD**.
 - **Las tareas se dividen y se ejecutan simultáneamente en varios procesadores** con diferentes entradas para obtener resultados más rápidamente.
 - **SPMD es el estilo más común de programación paralela.** También es un requisito previo para la investigación de conceptos como mensajes activos y memoria compartida distribuida.
- **Cada proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su "identidad".**
- **Las instancias interactúan llamando a funciones MPI,** que soportan comunicaciones proceso a proceso y grupales.

Funciones

- **MPI_init:**
 - **Inicializar entorno MPI.**
- **MPI_finalize:**
 - **Cerrar entorno MPI.**
- **MPI_common_size y MPI_common_rank:**
 - **Cantidad de procesos en el comunicador e identificador del proceso dentro del comunicador.**

Tipo de Datos MPI
 MPI_CHAR signed char
 MPI_SHORT signed short int
 MPI_INT signed int
 MPI_LONG signed long int
 MPI_UNSIGNED_CHAR unsigned char
 MPI_UNSIGNED_SHORT unsigned short int
 MPI_UNSIGNED unsigned int
 MPI_UNSIGNED_LONG unsigned long int
 MPI_FLOAT float
 MPI_DOUBLE double
 MPI_LONG_DOUBLE long double
 MPI_BYTE
 MPI_PACKED

Comunicaciones colectivas, para todos los procesos asociados a un comunicador, todos los procesos deben llamar a la rutina colectiva.
 MPI_Barrier
 MPI_Bcast
 MPI_Scatter - MPI_Scatterv
 MPI_Gather - MPI_Gatherv
 MPI_Reduce
 otras...

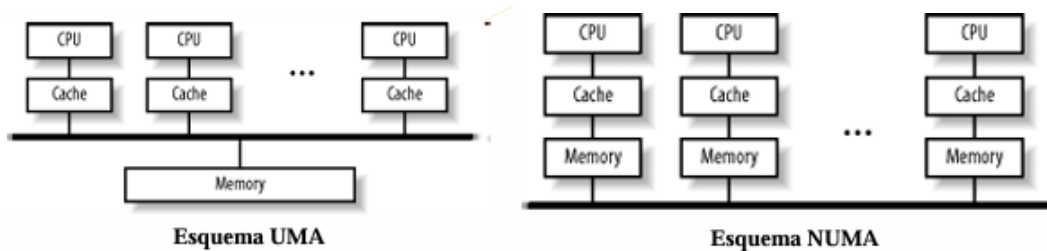
- **MPI_send y MPI_recv:**
 - Ambos **bloqueantes**.
- **MPI_Isend y MPI_Irecv:**
 - No **bloqueantes**.

Comunicadores

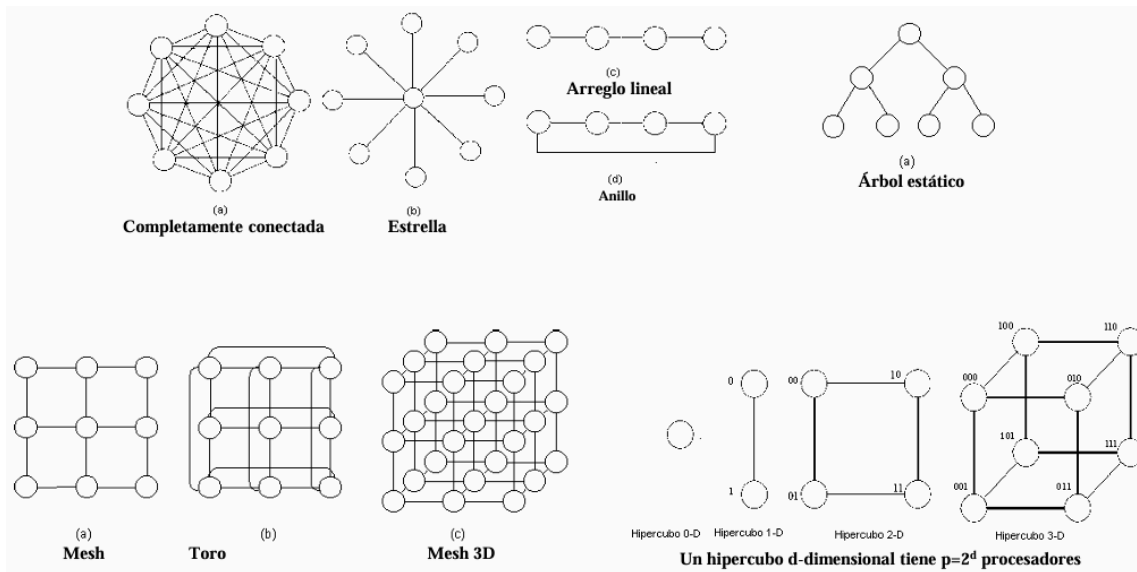
- Un comunicador **define el dominio de comunicación**.
- **Cada proceso puede pertenecer a muchos comunicadores**.
- Existe un **comunicador que incluye todos los procesos de la aplicación MPI_COMM_WORLD**.
- Son **variables del tipo MPI_Comm**
 - **Almacena información sobre qué procesos pertenecen a él.**
- En **cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar**.

Clasificación de Arquitecturas Paralelas

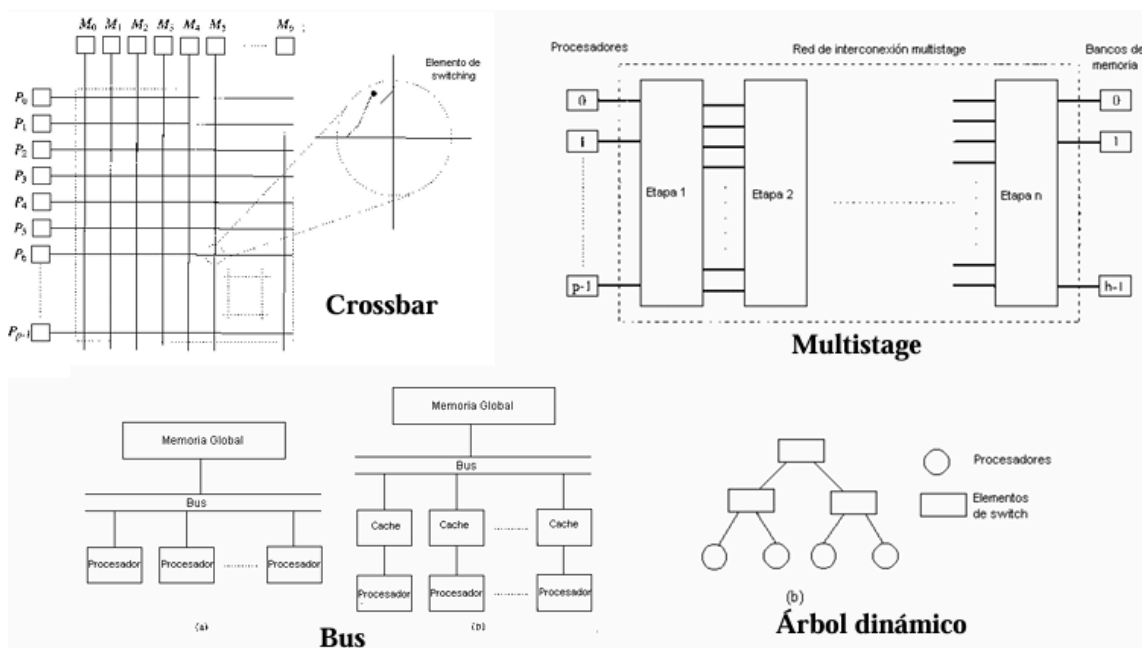
- **Por la organización del espacio de direcciones:** Se relaciona con el modelo de comunicación a utilizar (accesos a memoria o pasaje de mensajes).
 - **Memoria compartida:** Accesos a memoria compartida.
 - Esquemas **UMA**.
 - Esquemas **NUMA**.



- **Memoria distribuida:** Intercambio de mensajes.
- **Por la granularidad.**
- **Por el mecanismo de control:** Se basa en la manera en que las instrucciones son ejecutadas sobre datos
 - **SISD** (Single Instruction Single Data).
 - **SIMD** (Single Instruction Multiple Data).
 - **MISD** (Multiple Instruction Single Data).
 - **MIMD** (Multiple Instruction Multiple Data).
- **Por la red de interconexión:**
 - **Redes estaticas:** Constan de links **punto a punto**, típicamente se usan para maquinas de pasaje de mensajes.



- **Redes dinámicas:** Están **construidas usando switches y enlaces de comunicación.** Normalmente **para máquinas de memoria compartida.**



Arquitecturas **clasificadas por Mecanismos de Control**

SISD - Single Instruction Single Data

- **Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.** La memoria afectada es **usada sólo por esta instrucción.**
- Usada por la **mayoría de los uniprosesadores.**
- La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas.
- **Ejecución determinística.**

SIMD - Single Instruction Multiple Data

- Tiene **múltiples flujos de datos pero sólo un flujo de instrucción**.
- En particular, **cada procesador ejecuta exactamente la misma secuencia de instrucciones, y lo hacen en un lockstep**.
 - Esto hace a las máquinas SIMD **especialmente adecuadas para ejecutar algoritmos paralelos de datos**.
- Logra un **paralelismo a nivel de datos**, o sea ejecuta la misma instrucción en todos los procesadores.

MISD - Multiple Instruction Single Data

- Los procesadores ejecutan un **flujo de instrucciones distinto pero comparten datos comunes**.
- Operación sincrónica (en **lockstep**).
- No son máquinas de propósito general ("**hipotéticas**", Duncan).

MIMD - Multiple Instruction Multiple Data

- Cada **procesador tiene su propio flujo de instrucciones y de datos** entonces **cada uno ejecuta su propio programa**.
- Pueden ser **con memoria compartida o distribuida**.
- Logra **paralelismo total**, en cualquier momento se pueden estar ejecutando diferentes instrucciones con diferentes datos en procesadores diferentes.
- Ideal para simulaciones, comunicación y diseño.
- **Sub-clasificación de MIMD:**
 - **MPMD (Multiple Program Multiple Data):**
 - Cada procesador ejecuta su propio programa (ejemplo con PVM).
 - **SPMD (Single Program Multiple Data):**
 - Hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).

Diseño de Algoritmos Paralelos

- Para desarrollar un algoritmo paralelo el primer punto es **descomponer el problema en sus componentes funcionales concurrentes**(procesos/tareas).
 - Se trata de definir un **gran número de pequeñas tareas para obtener una descomposición de grano fino**, para brindar la **mayor flexibilidad** a los algoritmos paralelos potenciales.
 - En etapas posteriores se puede llegar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y **aglomerando tareas** para incrementar su tamaño o granularidad.
 - Esta descomposición puede realizarse de muchos modos. Un primer concepto es **pensar en tareas de igual código** (normalmente **paralelismo de datos o dominio**) pero también podemos tener diferente código (**paralelismo funcional**).
- Pasos Fundamentales: **Descomposición en Tareas y Mapeo de Procesos a Procesadores**.

Descomposición de Datos

- **Determinar una división de los datos y luego asociar el cómputo:**

- Esto da un número de tareas, donde **cada uno comprende algunos datos y un conjunto de operaciones sobre ellos**. Una operación puede requerir datos de varias tareas, y esto **llevará a la comunicación**.
- Son posibles distintas particiones, basadas en diferentes estructuras de datos.

Descomposición Funcional

- **Primero descompone el cómputo en tareas disjuntas y luego trata los datos.**
 - La descomposición funcional tiene un rol importante como **técnica de estructuración del programa, para reducir la complejidad del diseño general**.

Aglomeración

- El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que **no es especializado para ejecución eficiente en una máquina particular**.
- Esta etapa **revisa las decisiones tomadas** con la visión de **obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real**.
- **En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación.**

Objetivos que guían las decisiones de Aglomeración y Replicación

- **Incremento de la granularidad:**
 - **Intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola.**
- **Preservación de la flexibilidad:**
 - Al juntar tareas **puede limitarse la escalabilidad del algoritmo**. La capacidad para **crear un número variante de tareas es crítica si se busca un programa portable y escalable**.
- **Reducción de costos de IS (Ingeniería de Software):**
 - Se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

Características de las Tareas

- Una vez que tenemos el problema separado en tareas conceptualmente independientes, tenemos una serie de **características de las mismas que impactarán en la performance alcanzable** por el algoritmo paralelo:
 - **Generación** de las tareas.
 - **El tamaño** de las tareas.
 - **Conocimiento del tamaño** de las tareas.
 - **El volumen de datos asociado** con cada tarea.

Mapeo de Procesos a Procesadores

- Se **especifica dónde ejecuta cada tarea**.
- Este problema **no existe en uniprosesadores o máquinas de memoria compartida con scheduling de tareas automático**.
- **Objetivo:**
 - **Minimizar** el tiempo de ejecución. **Dos estrategias**, que a veces conflictúan:
 - **Ubicar tareas que pueden ejecutar concurrentemente en distintos procesadores** para mejorar la concurrencia. ✓

- Poner tareas que se comunican con frecuencia en iguales procesadores para incrementar la localidad.
- **El problema es NP-completo:**
 - No existe un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs entre estrategias en el caso general. Existen heurísticas para clases de problema.
- Normalmente tendremos más tareas que procesadores físicos.
- Los algoritmos paralelos (o el scheduler de ejecución) deben proveer un mecanismo de "mapping" entre tareas y procesadores físicos.
- Nuestro lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia/prioridad para el caso que no haya suficientes procesadores para atenderlas.
- La dependencia de tareas condicionarán el balance de carga entre procesadores.
- La interacción entre tareas debe tender a minimizar la comunicación de datos entre procesadores físicos.

Criterio para el Mapeo de Tareas a Procesadores

- **Criterios:**
 - Tratar de mapear tareas independientes a diferentes procesadores.
 - Asignar prioritariamente los procesadores disponibles a las tareas que estén en el camino crítico.
 - Asignar tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.
- Estos criterios pueden oponerse entre sí, el 3 puede llevar a no paralelizar.

Métrica de Speedup

- $S = T_s / T_p$
 - **S** es el cociente entre el tiempo de ejecución secuencial del algoritmo secuencial conocido más rápido (**T_s**) y el tiempo de ejecución paralelo del algoritmo elegido (**T_p**).
 - El rango de valores de **S** va desde 0 a p, siendo p el número de procesadores.
 - Mide cuánto más rápido es el algoritmo paralelo con respecto al algoritmo secuencial, es decir, cuánto se gana por usar más procesadores.

Métrica de Eficiencia

- $E = S / P$
 - Cociente entre speedup y número de procesadores.
 - El valor está entre 0 y 1, dependiendo de la efectividad en el uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.
 - Mide la fracción de tiempo en que los procesadores son útiles para el cómputo, es decir cuánto estoy usando de los recursos disponibles.

Programa Paralelizado

- En cualquier programa paralelizado existen dos tipos de código; el código paralelizado y el código secuencial.

- Existen ciertas secciones de código que ya sea por dependencias, por acceso a recursos únicos o por requerimientos del problema no pueden ser paralelizadas. Estas secciones conforman el código secuencial, que debe ser ejecutado por un solo elemento del procesador.
- Entonces, es lógico afirmar que la mejora del speedup de un programa dependerá del tiempo en el que se ejecuta el código secuencial, el tiempo en el que se ejecuta el código paralelizable y el número de operaciones ejecutadas de forma paralela.

Ley de Amdahl

- Enuncia que para cualquier tipo de problema existe un máximo speedup alcanzable que no depende de la cantidad de procesadores que se utilicen para resolverlo. Esto es así porque llega un momento en que no existe manera de aumentar el paralelismo de un programa.